Proyecto Fin de Máster en Ingeniería de Computadores
Curso académico 2009-2010

Estudio de rendimiento en GPU

Autor:
Carlos Juega Reimúndez

Directores del proyecto:
José Ignacio Gómez Pérez
Christian Tenllado Van der Reijden

Máster en Investigación en Informática
Facultad de Informática
Universidad Complutense de Madrid

Autorización

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: Estudio de rendimiento en GPU, realizado durante el curso académico 2009-2010 bajo la dirección de José Ignacio Gómez Pérez y Christian Tenllado Van der Reijden en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Carlos Juega Reimúndez

Resumen

En la actualidad las plataformas multicore lideran la industria de los computadores, obligando a los desarrolladores software a adaptarse a nuevos paradigmas de programación para poder explotar su capacidad de cómputo. A día de hoy uno de los principales exponentes de las plataformas multicore son las unidades de procesamiento gráfico (GPUs).

El desarrollo de aplicaciones para GPU requiere un alto esfuerzo por parte de los programadores. Por un lado, deben modelar los problemas de modo que permitan el aprovechamiento de plataformas masivamente paralelas. Por otro lado, deben preocuparse de que las aplicaciones hagan un uso eficiente del sistema de memoria, heterogéneo, de múltiples niveles, con gestión software o hardware. En general, dado un algoritmo concreto, el espacio de soluciones posibles para un mapeo sobre GPU es enorme. El recurso habitual de los programadores es el ensayo, prueba y error de distintas soluciones, guiados sólo por su propia experiencia e intuición, lo que resulta ineficiente de cara al desarrollo y mantenimiento de software.

En este proyecto hemos realizado un estudio sobre el impacto de distintas transformaciones de código de alto nivel en el rendimiento de distintos algoritmos en la GPU. Nuestro objetivo consiste en evaluar las distintas decisiones que debería tomar cualquier desarrollador al mapear un algoritmo sobre la GPU, identificando así aquellas que sean más importantes. Para ilustrar los resultados hemos utilizado como hilo conductor de la memoria la multiplicación de matrices. La extensión futura del trabajo consistirá en la definición de una metodología eficiente para el mapeo de aplicaciones sobre GPU.

Palabras clave: GPU, SIMT, CUDA, paralelización, optimización de rendimiento, modelo rendimiento, compiladores, multiplicación de matrices, convolución, NMF supervisado.

Abstract

At present, multicore platforms lead the computer industry, forcing software developers to adapt to new programming paradigms, in order to fully exploit their computing capabilities. Nowadays, graphics processing units (GPUs) are one of the main representatives of multi-core platforms.

The GPU application development requires a great effort by application programmers. On one hand, they must take advantage of massively parallel platform in the problem modeling. On the other hand, the applications have to make an efficient use of the memory system, which is heterogeneous, with several levels that are software or hardware controlled. Given a specific algorithm, the search space for the mapping is huge. Generally the programmers' methodology consists in evaluating several mapping alternatives, guided by their experience and intuition, which results inefficient for software development and maintenance.

This project deals with the study of the impact of several high level code transformations in the performance of different algorithms on the GPU. Our goal is the evaluation of the decisions that a programmer needs to make in the process of mapping an application on the GPU, identifying the most relevant. We use the matrix multiplication algorithm to illustrate our work. In the future this work will be completed by the definition of an efficient methodology for the mapping process.

Key words: GPU, SIMT, CUDA, parallelization, performance optimizations, performance model, compilers, matrix multiplication, convolution, supervised NMF.

Índice general

Índice de figuras

Capítulo 1

Introducción

Los microprocesadores basados en una única unidad de proceso (CPU), tales como la familia Pentium de Intel o la familia Opteron en AMD, incrementaron el rendimiento de las aplicaciones durante más de dos décadas. Estos microprocesadores alcanzaban varios gigaflops (GFLOPS) de cómputo en los equipos de escritorio y cientos de GFLOPS de cómputo en los servidores en cluster. Este implacable impulso en el rendimiento ha permitido que las aplicaciones software desarrollasen mayores funcionalidades, tuviesen mejores interfaces de usuario, etc. Los usuarios, en cambio, han ido demandando nuevas mejoras a medida que se han ido acostumbrando a las anteriores.

Durante este periodo, los desarrolladores software contaron con estas mejoras en el hardware para mejorar el rendimiento de sus aplicaciones de forma transparente; la misma aplicación simplemente se ejecutaba más rápido en cada nueva generación de microprocesadores. Sin embargo, este modelo de mejora del hardware se vió limitado a partir de 2003, ya que se alcanzaron altos niveles de consumos de energía que limitaron el aumento de la frecuencia de reloj y el nivel de actividad que podía realizarse en cada ciclo de reloj en una única CPU. Desde entonces, los fabricantes de microprocesadores han obtado por modelos de diseño multi-core y many-core, en los cuales existen varias unidades de proceso en el mismo chip, y así aumentar la capacidad de procesado. Esto ha provocado un enorme cambio entre los desarrolladores de software.

Tradicionalmente, la gran mayoría de aplicaciones software fueron desarrolladas siguiendo un modelo secuencial, tal y como describió Von Neumann en 1947. La ejecución de estos programas puede entenderse como el recorrido secuencial de su código. Históricamente, los usuarios estaban acostumbrados a que dichos programas se ejecutasen más

rápido en cada nueva generación de microprocesadores. Sin embargo, esto ya no es válido en nuestros días. Un programa secuencial se ejecutará en un único core, con lo que no se ejecutará más rápido de lo que se ejecuta hoy en día. Sin mejoras de rendimiento, los desarrolladores de software no pueden añadir nuevas características y capacidades en sus aplicaciones, reduciendo las opciones de crecimiento de toda la industria de la informática.

Por otro lado, las aplicaciones software que continuarán disfrutando de las mejoras de rendimiento con cada nueva generación de microprocesadores serán los programas paralelos, donde varios hilos de ejecución colaboran para conseguir acelerar la funcionalidad. Este hecho ha incentivado drásticamente el desarrollo de programas paralelos, en lo que se ha llegado a llamar la revolución del paralelismo. La práctica de la programación paralela no es en absoluto algo nuevo. En entornos de alto rendimiento se han desarrollado programas paralelos durante décadas. Sin embargo, el paradigma de la programación paralela quedaba reducida a un pequeño porcentaje de desarrolladores. Hoy en día todos los microprocesadores se basan en arquitecturas paralelas, y cada vez son más las aplicaciones desarrolladas siguiendo el paradigma de la programación paralela. Por ello, existe una gran necesidad, por parte de los desarrolladores de software, de aprender este modelo de programación.

Desde el 2003 y gracias en gran parte a la industria de los videojuegos, las tarjetas gráficas han ido evolucionando hasta convertirse en auténticos procesadores. El microprocesador de dichas tarjetas, también conocido como Unidad de Procesado Gráfico (GPU), ha liderado la carrera de rendimiento en lo que a operaciones en punto flotante se refiere, tal y como muestra la Figura 1.1.

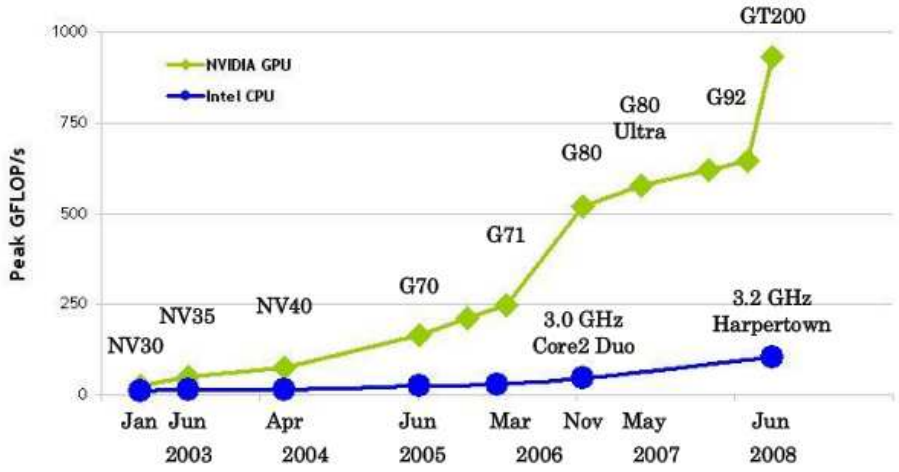


Figura 1.1: Comparativa rendimiento en GFLOPS entre CPUs y GPUs

La razón de esta diferencia en el redimiento entre CPU y GPU es que la GPU está especializada en computación intensiva y computación masivamente paralela, que es exactamente sobre lo que trata el renderizado de gráficos. Además, están diseñadas de forma que más transistores se destinan al procesado de datos en lugar de al almacenamiento de datos y control de flujo.

Con este nuevo tipo de arquitectura presente, aparecen nuevas dificultades para el desarrollador de software. Ahora, no sólo debe dominar la programación paralela tradicional, sino que debe aprender a desarrollar sus aplicaciones para estas nuevas arquitecturas especificas, masivamente paralelas, y así aprovechar toda la potencia que ofrecen las soluciones hardware de hoy en día. Aún más complicado es obtener soluciones óptimas, aprovechar las ventajas de la jerarquía de memoria, evitar y/o ocultar los cuellos de botella, etc. Todo esto puede conseguirse mediante ciertas transformaciónes de alto nivel en el código fuente. En este contexto se mueve el presente trabajo, que pretende realizar una exploración manual sobre técnicas de desarrollo en GPUs, para tratar de obtener una metodología de traducción de aplicaciones.

El Capítulo 2 explica la arquitectura de una GPU moderna, mostrando las unidades de ejecución y la gestión de la memoria. También presenta el modelo de ejecución para familiarizar al lector con su funcionamiento. El Capítulo 3 muestra el tipo de estrategias, desde un punto de vista general, que han de usarse para optimizar las aplicaciones. También indica el tipo de métricas usadas a lo largo del trabajo para evaluar las distintas opciones, así como un método basado en profiling para obtenerlas. El Capítulo 4 se centra en las aplicaciones estudiadas a fondo, y explica cómo encajan las estrategias explicadas en el Capítulo 3. Para finalizar, el Capítulo 5 analiza los resultados obtenidos; y el Capítulo 6 agrupa las conclusiones en base a los resultados y presenta futuras líneas de investigación.

Capítulo 2

Arquitectura moderna de GPU

La Figura 2.1 muestra la arquitectura típica de una GPU actual. Está compuesta por un número escalable de multiprocesadores paralelos (SMs). Estos multiprocesadores se agrupan de tres en tres (o de dos en dos en arquitecturas más antiguas) en lo que se llama Cluster de Procesado de Hilos (TPC). El número de SMs varía desde las arquitectuas más antiguas (1), hasta las más modernas y de mayor gama (30).

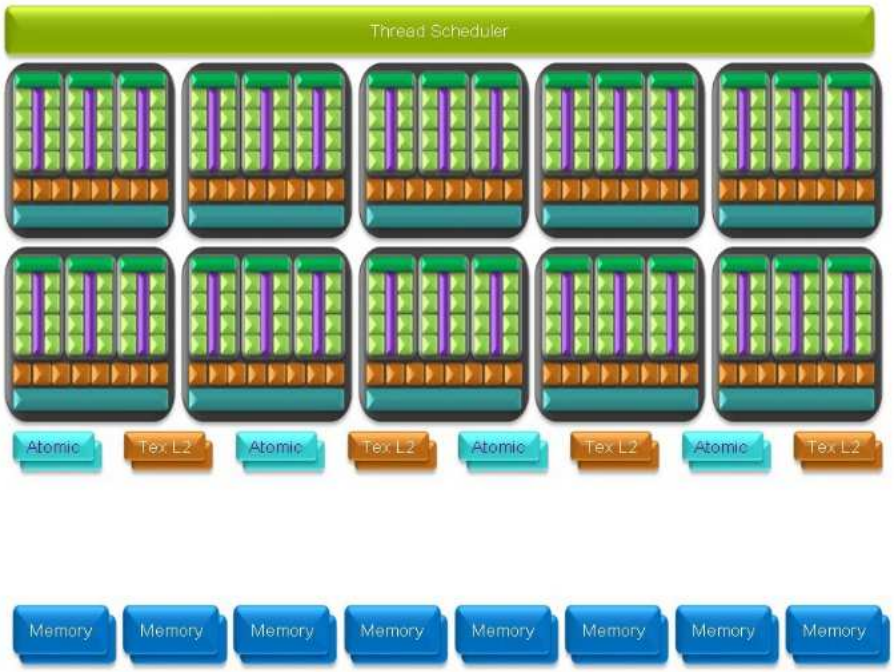


Figura 2.1: Arquitectura GPU moderna

El diseño interno de cada SM es similar para todas las versiones, cada SM cuenta con 8 procesadores escalares (SPs), lo que hace un total de 240 (30*8) procesadores en las tarjetas más modernas; 2 Unidades Especiales de Funcion (SFUs), capaces de realizar operaciones en punto flotante como SQRT y RCP SQRT, así como otras operaciones importantes. También cuenta con una unidad de multiplicación y suma (MAD) y una unidad adicional de multiplicación (MUL). Los ocho procesadores de un multiprocesador comparten la unidad de búsqueda y lanzamiento de instrucciones de forma que se ejecuta la misma instrucción al mismo tiempo en los ocho procesadores. Todas estas unidades funcionan a 1'35 gigahercios (GHz), esto son 933 GFLOPS de pico de cómputo.

Desde el punto de vista de la memoria, cada SM cuenta con tres módulos de memoria on-chip. La primera, una memoria compartida de lectura/escritura de 16KB, que ofrece un tiempo de acceso similar al de un registro. La segunda y tercera se corresponden con dos cachés de solo lectura: una de constantes y otra de texturas. Estos elementos se muestran en la Figura 2.2.

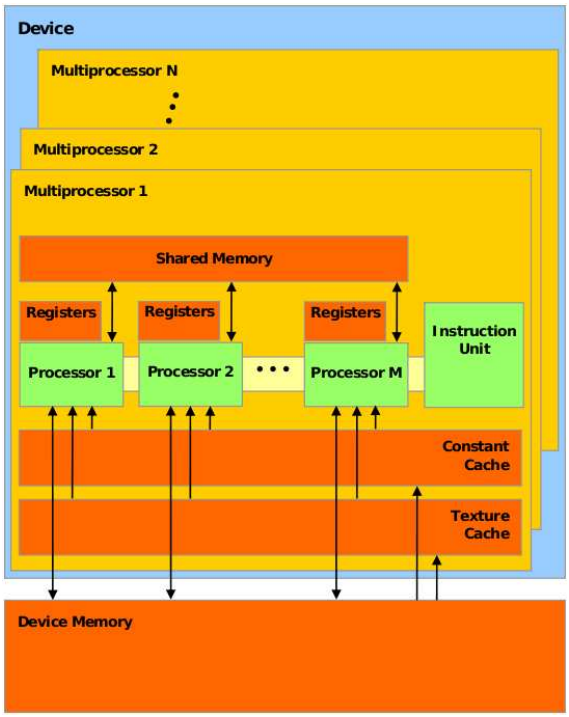


Figura 2.2: Arquitectura interna de un SM (Streaming Multiprocessor)

A nivel global, la tarjeta gráfica cuenta hasta con 4 gigabytes (GB) de memoria DRAM off-chip. En las aplicaciones gráficas, esta memoria almacena imagenes de video, texturas

para renderizados 3D, etc. Pero como procesador de proposito general, se comporta como una caché off-chip con un elevado ancho de banda (hasta 141 GB/s), aunque con una latencia superior a la caché convencional o el sistema de memoria. Si el chip se programa de forma adecuada, el elevado ancho de banda compensa esta mayor latencia en los accesos.

Actualmente la comunicación de la GPU con la CPU se realiza a través de un bus PCI-Express. Dicho bus consta de dos vías (una de envío y otra de recepción). El ancho de banda de cada vía es de 12'8 GB/s, lo que suma un total teórico de 25'5 GB/s para la comunicación con la CPU. Combinando el ancho de ambas vías se obtienen los 25'5 GB/s, pero en la práctica no se envían y reciben datos al mismo tiempo. Esta diferencia de ancho de banda entre la GPU-memoria GPU (141 GB/s) y GPU-memoria principal (12'8 GB/s) puede parecer una limitación, pero el ancho de banda PCI-Express es comparable al ancho de banda del Front-Side Bus (FSB) entre la CPU y el sistema de memoria principal, asi que en realidad no es tal limitación.

El chip G200 (NVIDIA) es masivamente paralelo con 240 procesadores. Una aplicación bien desarrollada permitirá que se ejecuten entre 25000 y 30000 hilos de forma simultánea en el chip. Nótese que los microprocesadores de Intel/AMD soportan de 2 a 4 hilos por core (16 hilos simultáneos en un procesador Quad-core), dependiendo de la arquitectura. La familia G200 soporta hasta 1024 hilos por cada multiprocesador, con sus 30 multiprocesadores suman un total de 30720 hilos simultáneos. Es muy importante comprender este concepto para poder escribir programas de forma eficiente.

2.1. Modelo de programación CUDA

El modelo de programación CUDA asume que los hilos CUDA se ejecutan en una unidad física distinta que actúa como coprocesador (*device*) al procesador (*host*) donde se ejecuta el programa (Figura 2.3). CUDA C es una extensión del lenguaje de programación C, que permite al programador definir funciones C, llamadas *kernels*, que, al ser llamadas, se ejecutan en paralelo por *N hilos* diferentes. Los kernels se ejecutan de forma secuencial en el *device*[1].

[1] Con la llegada de la nueva generación de GPUs, la arquitectura *Fermi* (chip GF100), es posible ejecutar kernels en paralelo

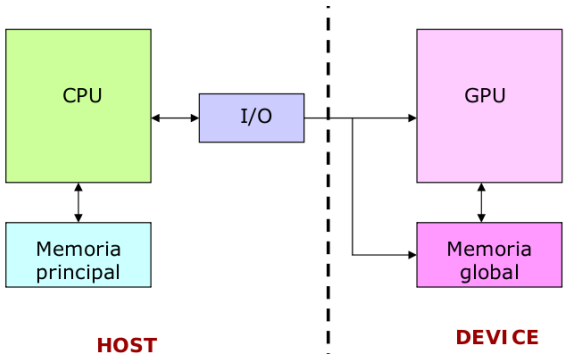


Figura 2.3: Arquitectura CPU-GPU

Como ejemplo, el siguiente código muestra cómo se define un kernel y cómo se llaman desde un programa:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

Existe una jerarquía perfectamente definida sobre los hilos de CUDA. Los hilos se agrupan en vectores a los que se les llama *bloques*, estos vectores pueden ser de una, dos o tres dimensiones, de forma que definen bloques de hilos de una, dos o tres dimensiones. Hilos del mismo bloque pueden cooperar entre si, compartiendo datos y sincronizando sus ejecuciones. Sin embargo, hilos de distintos bloques no pueden cooperar. Los bloques a su vez, se organizan en un *grid* de bloques. Este grid, de nuevo puede ser de una o dos dimensiones. Los valores entre <<< ... >>> que aparecen en código anterior se conocen como la configuración del kernel, y definen la dimensión del *grid* y el número de *hilos* de cada bloque. La Figura 2.4 muestra como se organizan los hilos en un grid de 2x3 bloques y 3x4 hilos cada uno.

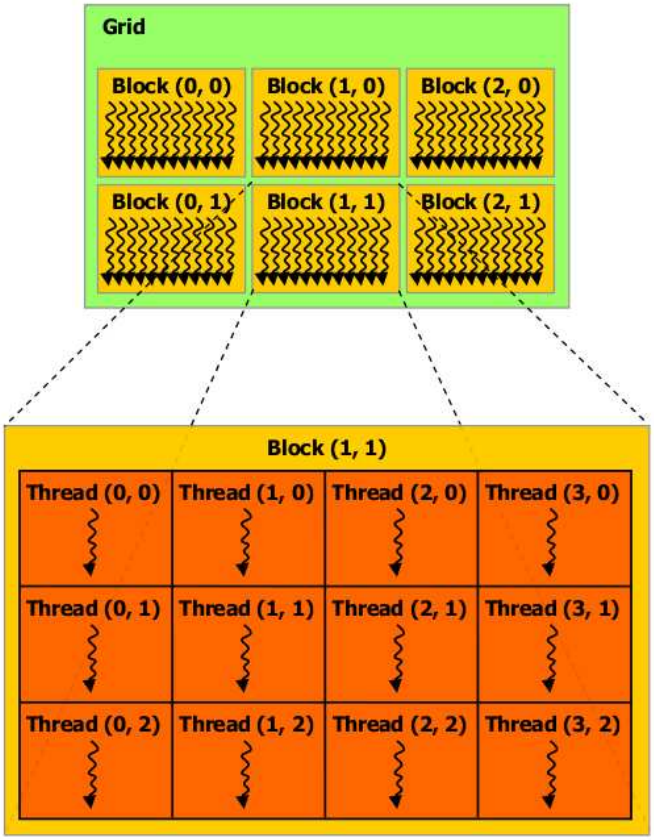


Figura 2.4: Jerarquía de hilos

Como puede verse, cada hilo queda perfectamente identificado por un *ID* de bloque y el *ID* del propio hilo dentro del bloque. Estos IDs suelen usarse como índices para definir qué porciones de los datos procesa cada hilo. Esto puede verse en el código anterior.

Cada hilo tiene acceso a distintos tipos de memoria. En primer lugar una *memoria local* al propio hilo, esta memoria se aloja en la memoria principal de la GPU (off-chip). Además todos los hilos de un mismo bloque comparten una región de *memoria compartida* (on-chip) para comunicarse entre ellos, la memoria compartida tiene el mismo tiempo de vida que el bloque de hilos. Por último, todos los hilos tienen acceso a la misma memoria global (*device memory*).

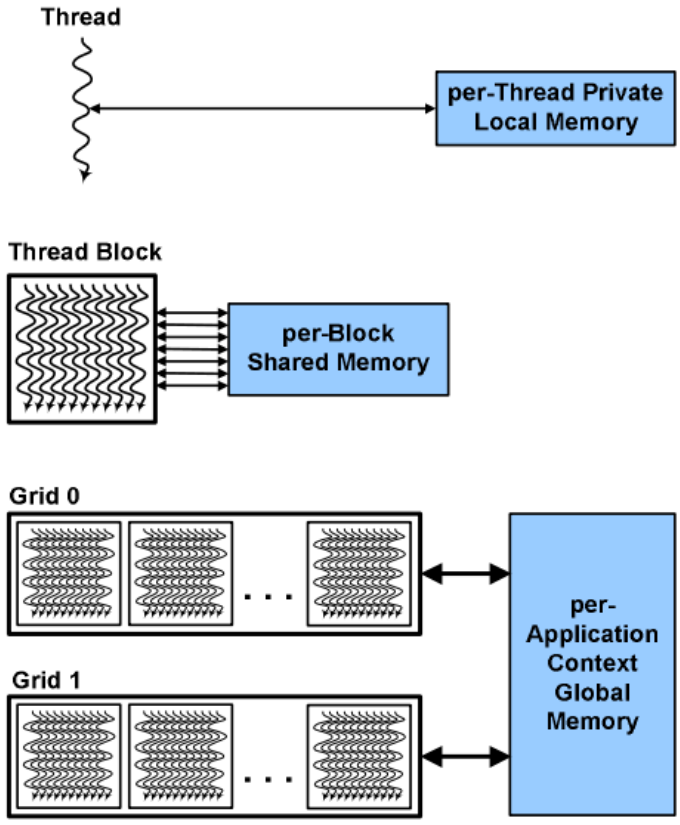


Figura 2.5: Jerarquía de memoria

También existen dos espacios de memoria de sólo lectura adicionales, accesible por todos los hilos, el espacio de memoria de constantes y el espacio de memoria de texturas, ambos optimizados para distintos usos. Los espacios de memoria global, memoria de constantes y memoria de texturas son persistentes a las llamadas entre distintos kernels.

2.2. Modelo de ejecución

La ejecucion de los hilos en la GPU no se lleva a cabo de forma independiente. El planificador de hilos mostrado en la Figura 2.1 planifica y distribuye bloques de hilos entre los SM. Cada SM puede ejecutar hasta ocho bloques de forma simultánea, siempre que se cumplan todas las restricciones sobre los recursos del multiprocesador. Una aplicación común ejecuta más de 240 bloques (30*8), por lo tanto, es tarea del planificador mantener una lista de los bloques planificados e ir asignando bloques a los SM según terminen.

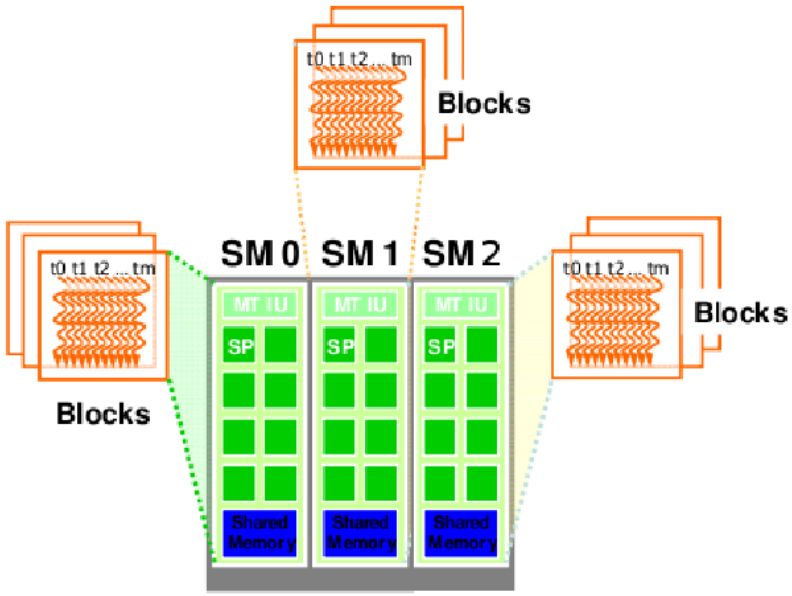


Figura 2.6: Asignación de bloques a los SMs (Streaming Multiprocessor) de un TPC

El multiprocesador crea, gestiona y ejecuta hilos concurrentes en el hardware sin sobrecoste de planificación o cambios de contexto. Mapea cada hilo a un procesador, y cada hilo se ejecuta de forma independiente con su propia dirección de instruccion y registros de estado. Este nuevo modelo de ejecución se ha llamado SIMT (*Single Instruction Multiple Thread*). En primer lugar, el multiprocesador divide los bloques de hilos en grupos de 32 hilos llamados *warp*. A la hora de lanzar una nueva instrucción, la unidad de planificación, selecciona un warp disponible y lanza la misma instrucción para todos los hilos de ese warp. Las instruciones de salto suponen un problema ya que hilos de un mismo warp pueden tomar caminos de ejecución distintos. En este caso, la ejecución se serializa, ejecutando primero los hilos de un camino y después los hilos del otro. Además existen instrucciones para sincronizar todos los hilos de un mismo bloque[2], haciendo que warps enteros detengan su ejecución hasta que todos los warps del bloque alcancen el mismo punto de ejecución.

[2]Nótese que la sincronización es a nivel de hilos de un mismo bloque, ya que los bloques de hilos son independientes entre si y su sincronización no es posible

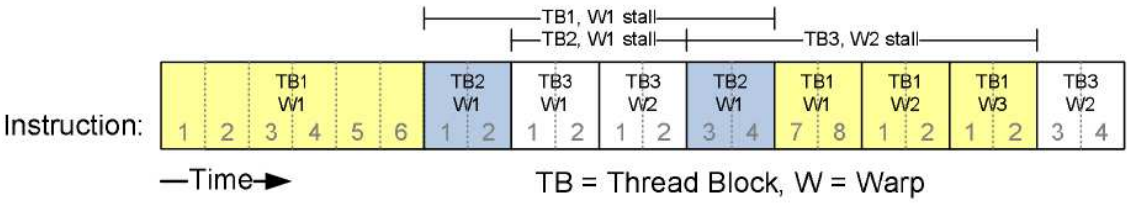


Figura 2.7: Ejemplo de ejecución de warps

La arquitectura SIMT es similar a la arquitectura vectorial SIMD (*Single Instruction Multiple Data*) en el sentido en que una instrucción controla varios elementos de procesado. Sin embargo, una diferencia clave es que la organización de los vectores SIMD exponen el ancho del vector al software, mientras que desde el punto de vista SIMT, las instrucciones especifican la ejecución y comportamiento de un único hilo. A diferencias de las máquinas SIMD, SIMT permite al programador describir paralelismo a nivel de hilo para hilos independientes, así como paralelismo a nivel de datos para hilos coordinados. El programador puede ignorar el comportamiento SIMT para el correcto funcionamiento de su código, sin embargo es un detalle muy importante para el rendimiento.

2.2.1. Acceso a memoria

En la Figura 2.7 se puede ver la ejecución de un kernel a lo largo del tiempo. El kernel tiene tres bloques (*TB1*, *TB2* y *TB3*) con al menos tres warps cada uno (*W1*, *W2* y *W3*). El planificador decide ejecutar el W1 de TB1 en primer lugar. Todos los hilos de ese warp ejecutan séis instrucciones y se producen un cambio de contexto. En ese momento, el planificador decide ejecutar el W1 de TB2. Los cambios de contexto se producen cuando se ejecuta una instrucción de memoria, y así evitar que el multiprocesador esté parado. La latencia de una instrucción de memoria varía dependiendo del espacio de memoria al que se accede, e incluso del patrón de acceso, tal y como se explica a continuación.

Como se ha visto en apartados anteriores (Figura 2.5), la arquitectura de la tarjeta gráfica envuelve distintos niveles de memoria, algunos de ellos on-chip (dentro de la GPU) y otros off-chip (fuera de la GPU). Desde el punto de vista de la ejecución, las instrucciones a memoria se tratan de forma diferente según a qué nivel se esté accediendo.

2.2.1.1. Memoria compartida

Como la *memoria compartida* está dentro del chip, es una memoria mucho más rápida que el espacio de *memoria local* y *global*. De hecho, el tiempo de acceso de todos los hilos de un warp accediendo a la memoria compartida es tan rápido como el acceso a los registros, siempre que no existan conflictos. La memoria compartida está diseñada en forma de 16 bancos de 1Kb en los que los datos se distribuyen a nivel de palabra. El acceso a bancos distintos se puede realizar de forma simultánea. Si se accede a datos que están en el mismo banco, entonces se produce un conflicto y el acceso se serializa. La serialización se lleva a cabo separando la petición a memoria en tantas peticiones como sean necesarias para que no existan conflictos, disminuyendo así el ancho de banda efectivo en un factor igual al número de peticiones libres de conflicto.

Entonces para obtener el máximo rendimiento, es necesario comprender como se mapean los datos en los bancos de la memoria compartida para tratar de minimizarlos. En la memoria compartida, los bancos están organizados de forma que sucesivas palabras de 32 bits se asignan a sucesivos bancos de memoria. Teniendo en cuenta que las arquitecturas modernas tienen 32 hilos por warp, siempre habría, al menos, conflicto de grado 2 si las peticiones se hiciesen por warp entero. Para evitar esto, cuando un warp ejecuta una instrucción sobre la memoria compartida, la petición se separa en dos peticiones: una para la primera mitad del warp y otra para la segunda mitad. De este modo cada uno de los 16 hilos del medio warp puede acceder a un banco y obtener el máximo rendimiento. A continuación se muestran algunos ejemplos de patrones de acceso que no provocan conflictos en memoria compartida:

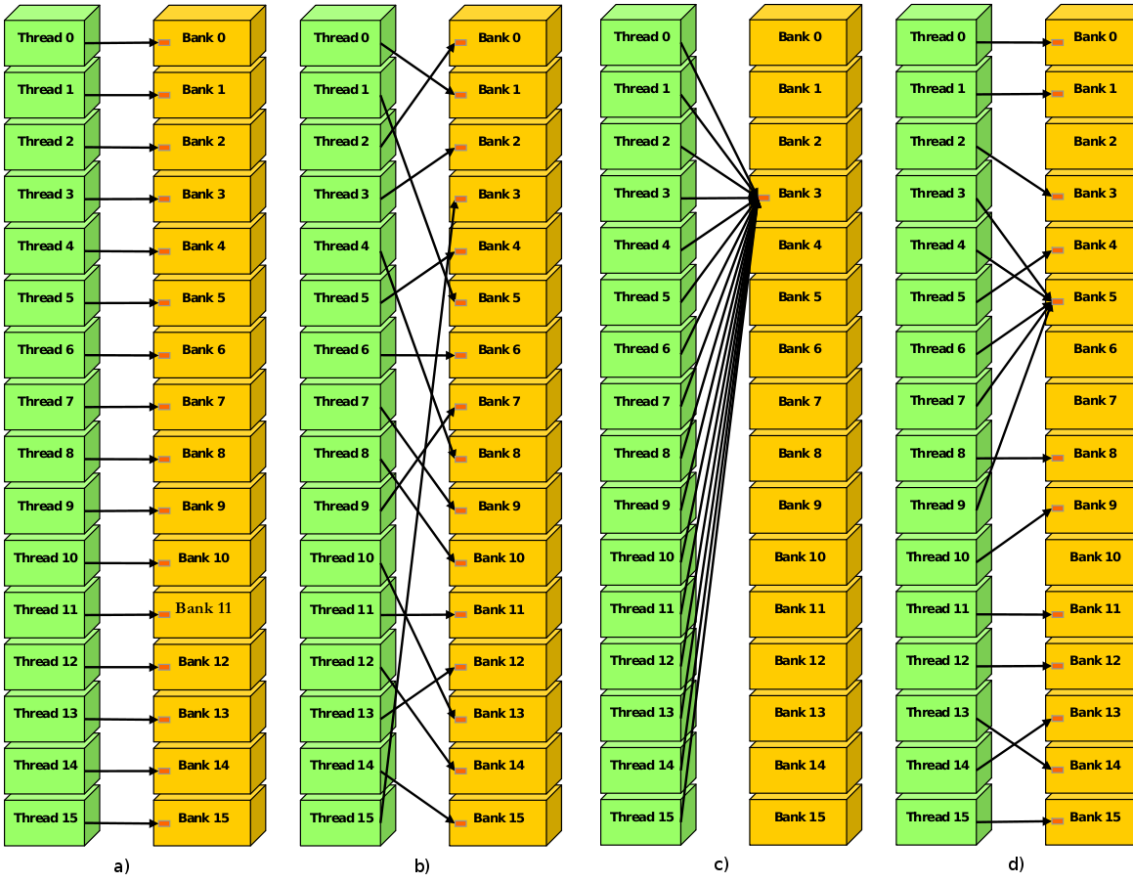


Figura 2.8: Patrones de acceso que no causan conflicto en memoria compartida

En la Figura 2.8 los patrones *a)*, acceso lineal de los hilos a palabras de 32 bits, y *b)*,permutación aleatoria, no provocan ningún conflicto ya que cada hilo accede a un banco distinto. Además, la memoria compartida también implementa un mecanismo de distribución por el cual una palabra de 32 bits se puede leer por varios hilos simultáneamente en la misma petición de lectura. Esto reduce el número de conflictos sobre un banco al que le piden el mismo dato varios hilos. La Figura 2.8 en sus casos *c)* y *d)* muestra esta situación.

Como contrapartida, la Figura 2.9 muestra un par de ejemplos en los que el patrón de acceso a la memoria global provoca conflictos.

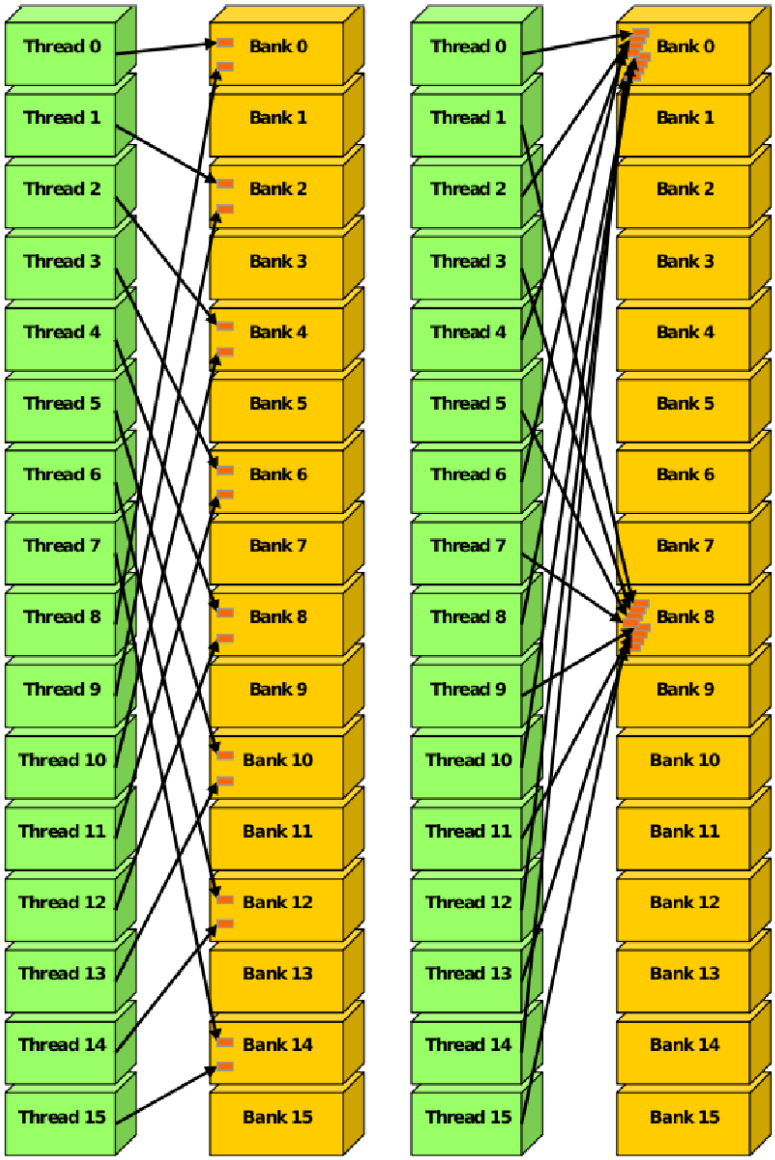


Figura 2.9: Patrones de acceso que causan conflicto en memoria compartida

El patrón de acceso de la izquierda muestra conflictos de grado 2 al acceder a los datos con un desplazamiento de dos palabras de 32 bits. En la derecha se pueden ver conflictos de grado 8 al acceder a los datos con un desplazamiento de ocho palabras de 32 bits.

2.2.1.2. Memoria global

La *memoria global* es mucho más lenta que la *memoria compartida* ya que se encuentra fuera del chip y por lo tanto es mucho más importante realizar las peticiones de forma eficiente. La memoria global (*device memory*) se divide en tres espacios de memoria separados: espacio de memoria global, espacio de memoria de texturas y espacio de memoria de constantes. Los dos primeros son espacios de memoria accesibles tanto lectura como escritura; sin embargo, la memoria de constantes es un espacio dedicado únicamente a la lectura. Además los espacios de texturas y constantes cuentan con cachés on-chip como se mostró en la Figura 2.2.

Al igual que en los accesos a la memoria compartida, los accesos al espacio de memoria global pueden tener diferentes latencias dependiendo del patrón de acceso. Las peticiones a memoria global también las hacen medio warp. Por lo tanto, cuando una instrucción a memoria global es ejecutada por un warp, en realidad se hacen dos peticiones: una para la primera mitad del warp y otra para la segunda mitad. Para aumentar la eficiencia de la memoria global, el hardware puede unificar las transacciones dependiendo del patrón de acceso. Las restricciones para la unificación depende de la arquitectura, en las más modernas basta con que todos los hilos de medio warp accedean al mismo segmento de memoria (las restricciones de las arquitecturas más antiguas pueden encontrarse en [7]). El patrón de acceso dentro del segmento no importa, varios hilos pueden acceder a un dato, puede haber permutaciones, etc. Sin embargo, si los hilos acceden a n segmentos distintos de memoria, entonces se producen n transacciones. El tamaño del segmento ha de ser:

- 32 bytes si todos los hilos acceden a palabras de 1 byte,
- 64 bytes si todos los hilos acceden a palabras de 2 bytes,
- 128 bytes si todos los hilos acceden a palabras de 4 bytes.

Si los hilos no acceden a todos los datos del segmento, entonces se leen datos que no serán usados y se despercidia ancho de banda. Por ello, el hardware facilita un sistema que acota la cantidad de datos a traer dentro del segmento, pudiendo traer subsegmentos de 32 bytes o 64 bytes. La Figura 2.10 muestra algunos ejemplos de acceso a la memoria global.

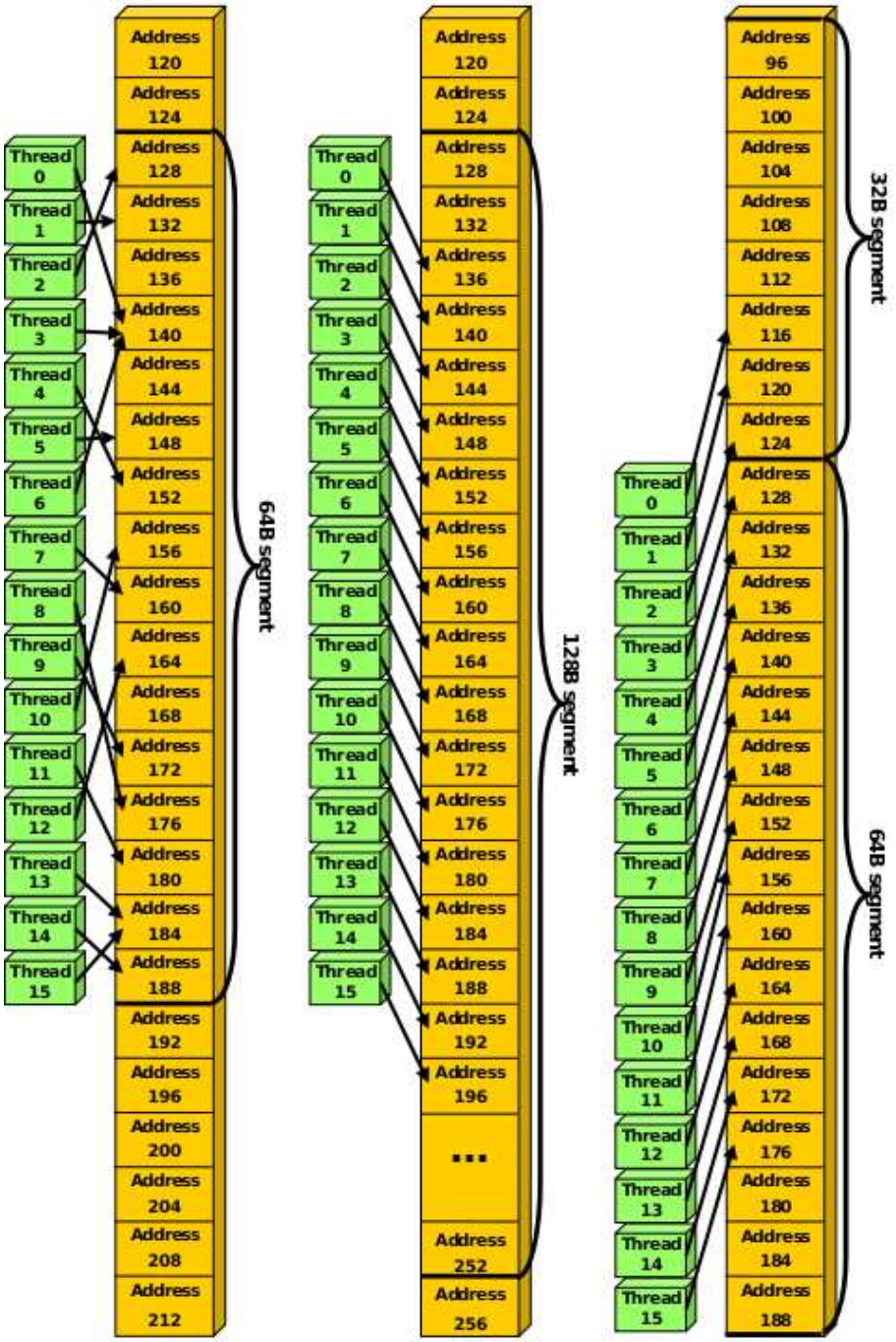


Figura 2.10: Patrones de acceso a la memoria global

En los tres casos los hilos acceden a palabras de 4B, así que el tamaño de segmento es de 128B. En el caso de la izquierda, los hilos acceden a 16 posiciones consecutivas

alineadas con el segmento de 128B y el hardware reduce el tamaño a 64B para evitar leer datos inútiles. De esta forma, realiza una única transacción de 64B. En el centro ocurre precisamente lo contrario. También se acceden a 16 posiciones, pero al no estar alineadas, el hardware no puede reducir la cantidad de datos a leer y genera una transacción de 128B. Por último, en la derecha, los hilos acceden a palabras alojadas en distintos segmentos de 128B; y no queda más remedio que generar dos transacciones, aunque como puede verse, el tamaño de las transacción se reduce a 32B y 64B.

Memoria local

El espacio de *memoria local* se encuentra dentro del espacio de *memoria global* y por lo tanto, es igual de costoso acceder a ella. La memoria local se utiliza de forma automática por el compilador al alojar variables que no caben en registros o que elevan mucho el número de registros usados por cada hilo. El *spilling* de registros se aloja en esta memoria.

Memoria constante

El espacio de *memoria constante* está cacheado. Por lo tanto, una lectura es igual de costoso que una lectura en *memoria global* sólo si se produce un fallo en caché. En caso de acierto, el coste de acceder a los datos en la caché de constantes es variable: siodos los hilos de medio warp acceden a la misma dirección de la caché, el coste es igual que acceder a los registros. Este coste aumenta de forma lineal con el número de direcciones a las que acceden los hilos.

Memoria de texturas

El espacio de *memoria de texturas* también está cacheado, asique, al igual que la *memoria constante*, sólo accede a la *memoria global* si se produce un fallo en caché. Sin embargo, en caso de acierto el coste es distinto. La *memoria de texturas* está optimizada para aprovechar la localidad espacial de dos dimensiones. De esta forma, se consigue mejor rendimiento a medida que los hilos del mismo warp acceden a direcciones más cercanas de la memoria.

Realizar las lecturas a través de la *memoria de texturas* puede tener algunos beneficios que la conviertan en una mejor alternativa a la *memoria global* o la *memoria constante*:

- Si las lecturas no se ajustan a los patrones de acceso a la memoria global o a la memoria de constantes explicados anteriormente, es posible obtener mayor ancho de banda explotando las ventajas de localidad en la memoria de texturas.
- La latencia debida al cálculo de direcciones se oculta mejor y posiblemente mejore el rendimiento de las aplicaciones que acceden a los datos de forma aleatoria.
- Los datos pueden se distribuidos a variables separadas en una única instrucción.
- Los enteros de 8 bits y 16 bits pueden ser convertidos a punto flotante de 32 bits (en rangos [0.0, 1.0] o [-1.0, 1.0]).

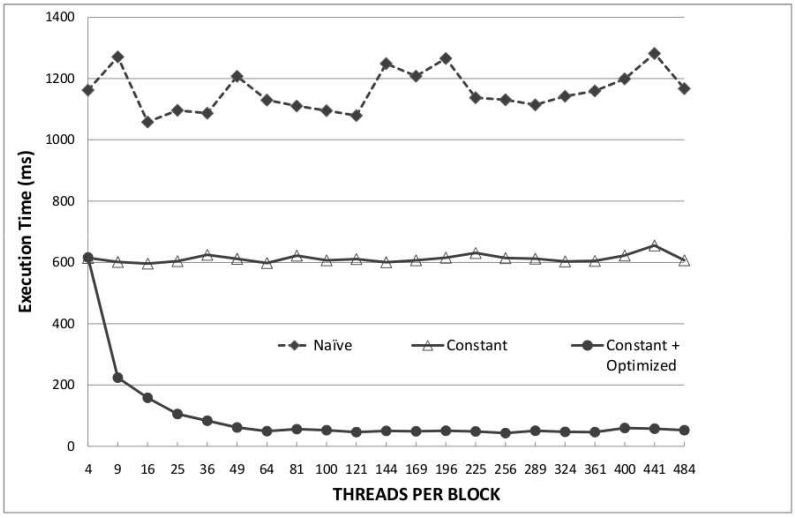
Capítulo 3

Estrategias y métricas

Como se explicó en el Capítulo 1, este proyecto pretende presentar un estudio de rendimiento de diferentes aplicaciones en la tarjeta gráfica. Inicialmente servirá para acercar este nuevo modelo de programación a los desarrolladores que no estén familiarizados con el entorno. Sin embargo, el objetivo a largo plazo es realizar la traducción automática de código C a código CUDA, y que el código resultante sea eficiente. Para ello, es conveniente obtener previamente un modelo de rendimiento de la GPU. Dicho modelo debe ser capaz de predecir el impacto que tienen sobre el rendimiento distintas transformaciones de alto nivel.

Para justificar este trabajo, en la Figura 3.1 se muestran dos gráficas. La primera corresponde al rendimiento de diferentes implementaciones del kernel SVM, usado en algoritmos de clasificación de carass. Hay tres implementaciones del SVM: *naive*, *constant* y *constant+optimized*. *Naive* usa únicamente memoria global, *constant* utiliza memoria constante y *constant+optimized*, además de usar memoria constante, reordena las lecturas para minimizar el número de transacciones a memoria global.

La segunda corresponde a la multiplicación de matrices, algoritmo que servirá de ejemplo conductor del trabajo. En la gráfica se muestra la variación del tiempo de ejecución frente a distintas configuraciones de una implementación del algoritmo (número de bloques, hilos por bloque, etc.), pero manteniendo constante el número de recursos de los multiprocesadores.



Distintas configuraciones en la multiplicación de matrices

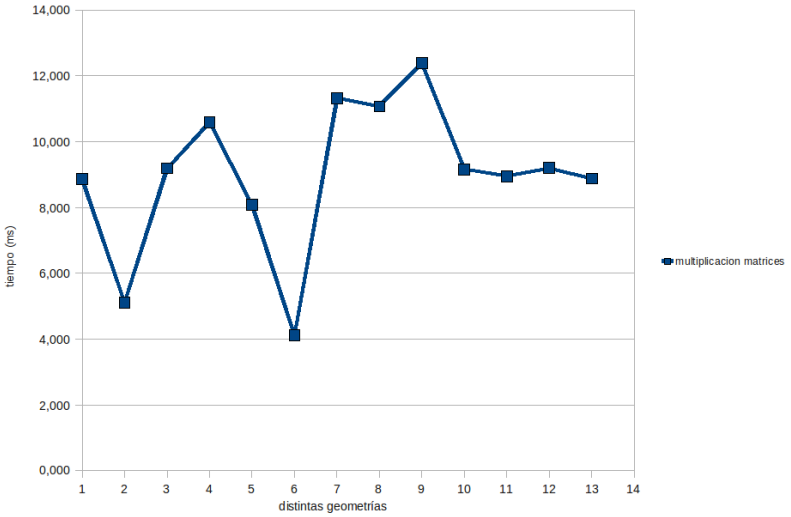


Figura 3.1: Ejemplo motivacional sobre el rendimiento

Como puede verse en la Figura 3.1, el tiempo de ejecución varía notablemente dependiendo de la implementación y configuración. Esto justifica la necesidad, por parte de los desarrolladores, de optimizar su código. Y desde el punto de vista de este trabajo, el estudio exhaustivo del rendimiento para tratar de encontrar un método eficaz de traducción. A grandes rasgos, las tres lineas a seguir para optimizar los kernels son:

- Optimizar el uso de la memoria para obtener el máximo ancho de banda en cada

nivel de memoria.

- Maximizar la ejecución paralela.
- Optimizar el flujo de instrucciones para obtener la máxima tasa de instrucciones ejecutadas, elevar el paralelismo a nivel de instrucción, etc.

Normalmente, no es posible optimizar los tres objetivos a la vez, ya que suelen entrar en conflicto unos con otros. Entonces hay que llegar a un compromiso entre ellos para sacar ventaja en conjunto. En primer lugar, es muy importante dar trabajo a todos los multiprocesadores. Como se explicó en el Capítulo 2, los bloques se van distribuyendo por todos los multiprocesadores. Erróneamente se puede pensar que basta con que haya un bloque por multiprocesador; en tal caso, las instrucciones de sincronización y los tiempos de espera por instrucciones a memoria pueden provocar que el multiprocesador se quede parado. Por ello, es mejor configurar los kernels con un número elevado de bloques. De esta forma, aunque solo un máximo de ocho bloques se ejecuten a la vez, el multiprocesador podrá planificar otros bloques en caso de paradas en la ejecución de los bloques activos. Basándonos en nuestros resultados, una buena relación se alcanza cuando al menos se asignan 30 bloques a cada multiprocesador.

Aún así, el espacio de decisiones es tan grande (niveles de memoria, paralelismo, geometría de bloques, asignación de trabajo a los hilos, ...), y las decisiones dependen tanto del tipo de aplicación a traducir (véase intensiva en memoria, intensiva en computación, irregular, etc); que imposibilita la búsqueda de soluciones óptimas de forma manual. Sin embargo, para tratar de automatizar el proceso es necesaria una exploración manual para encontrar una metodología. Por ello inicialmente exploraremos distintas alternativas de forma manual. Basándonos en los resultados de la Figura 3.1, nuestra exploración manual comenzará por las optimizaciones de memoria, ya que en la Figura 3.1 las optimizaciones en la jerarquía de memoria producen cambios más pronunciados en el rendimiento.

En general, y teniendo en cuenta que el principal cuello de botella de la GPU es el sistema de memoria, estas optimizaciones han de ser las que mejor resultado obtengan en cualquier tipo de aplicación, aunque ésta hago poco uso de él. Pues los accesos a memoria suponen muchos ciclos de reloj desaprovechados.

3.1. Optimizar uso de memoria

A lo largo de la Sección 2.2.1 se explicó el funcionamiento del sistema de memoria de la GPU. Cúando hacer uso de cada nivel de memoria depende íntegramente del algoritmo. Por ejemplo, si un algoritmo tiene unos datos de entrada a los que acceden todos los hilos a la vez, tiene sentido que esos datos estén en el espacio de memoria constante. Si por otro lado, los datos a acceder cambian a lo largo del algoritmo, esos datos deben mapearse en el espacio de memoria global y/o compartida. El Cuadro 3.1 muestra las distintas formas de declarar variables en CUDA, el alcance, el tiempo de vida y su correspondiente mapeo en los distintos niveles de memoria.

Declaración de variable	**Memoria**	**Alcance**	**Tiempo de vida**
variables excepto arrays	registros	hilo	kernel
arrays	global	hilo	kernel
__shared__ int sharedVar;	compartida	bloque	kernel
__device__ int globalVar;	global	grid	aplicación
__constant__ int constVar;	constante	grid	aplicación

Cuadro 3.1: Variables CUDA y memoria

El Cuadro 3.1 muestra un hecho de gran importancia: las variables se mapean a registros pero los arrays se mapean a memoria local (que forma parte de la memoria global). Esto, que en las arquitecturas convencionales no tiene ninguna implicación, en la GPU supone que, a alto nivel, el uso de variales escalares es más eficiente que el uso de arrays. Aún así, los datos se suelen alojar en el espacio de memoria global, y por lo tanto, nuestras primeras estrategias de optimización se basarán en acceder a este nivel de memoria.

3.1.1. Explotando localidad

Existe un compromiso intrínseco en el uso de las distintas memorias en CUDA: la memoria global es grande pero lenta, mientras que la memoria compartida es pequeña pero rápida. Por tanto es imprescindible explotar la localidad temporal y espacial, pero adaptadas en la GPU. Realizar un *tiling* permite alojar partes de una matriz en la memoria compartida. El objetivo es reducir al mínimo el número de accesos a la memoria global y convertir el resto de accesos en accesos a memoria compartida.

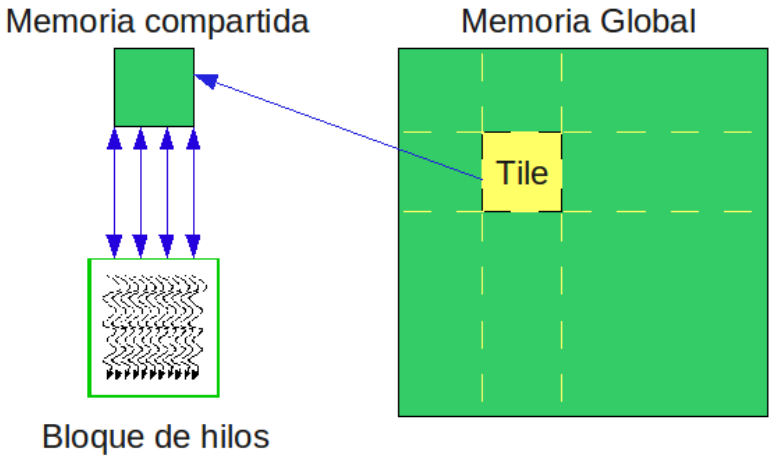


Figura 3.2: Tiling sobre la memoria compartida

En la Figura 3.2, es sencillo comprobar que a medida que aumenta el tamaño del tile, disminuye el número de accesos a memoria global a razón:

$$cantidad\_datos\_en\_global/tamano\_tile \tag{3.1}$$

Sin embargo, aumentar el tamaño de los tiles puede mermar el factor de paralelismo. Cada multiprocesador cuenta con 16KB de memoria compartida (Sección 2.2.1.1). Si se asigna mucha memoria compartida por bloque, el número de bloques activos (se ejecutan simultáneamente) puede no ser máximo. Esto se traduce en perdida de paralelismo. En caso que no existan otras limitaciones, el Cuadro 3.2 muestra el máximo número de bloques de hilos activos en función de la cantidad de memoria compartida asignada a cada bloque.

Memoria compartida (Bytes)	**Bloques**
0 - 2048	8
2048 - 2560	6
2560 - 3072	5
3072 - 4096	4
4096 - 5120	3
5120 - 8192	2
8192 - 16384	1

Cuadro 3.2: Limite de bloques activos por multiprocesador según la memoria compartida

Además de definir un tamaño adecuado para la memoria compartida, también hay que

definir qué datos deben ir a la memoria compartida de forma que se explote la localidad y el reuso de los datos. Una vez más, los datos propensos a ser enviados a la memoria compartida son los más accedidos. No tiene sentido por ejemplo llevar a memoria compartida datos a los que solo se accede una vez, pues esto supondría acceder dos accesos (memoria global + memoria compartida) a memoria en lugar de uno (solo memoria global). Sin embargo, si merece la pena llevar un dato a memoria compartida en caso que varios hilos acceden una o más veces a un mismo dato a lo largo de su ejecución.

Por último, para evitar conflictos en los bancos de la memoria compartida hay que evitar que hilos contiguos (mismo medio warp) accedan al mismo banco de memoria. En la Sección 2.2.1.1 explicamos que los datos se distribuyen en los bancos a nivel de palabra; por lo tanto, los conflictos ocurriran cuando hilos contiguos acceden a direcciones tales que $dir\,\%16$ coinciden. En esta situación, hay que tratar de cambiar el patrón de acceso a la memoria compartida, pero si no es posible, es preferible sacrificar un poco de memoria compartida para evitar la condición anterior. De este modo se malgasta un poco de memoria pero se garantizan que los accesos no tienen que serializarse. Esta técnica se conoce como *padding* sobre memoria compartida.

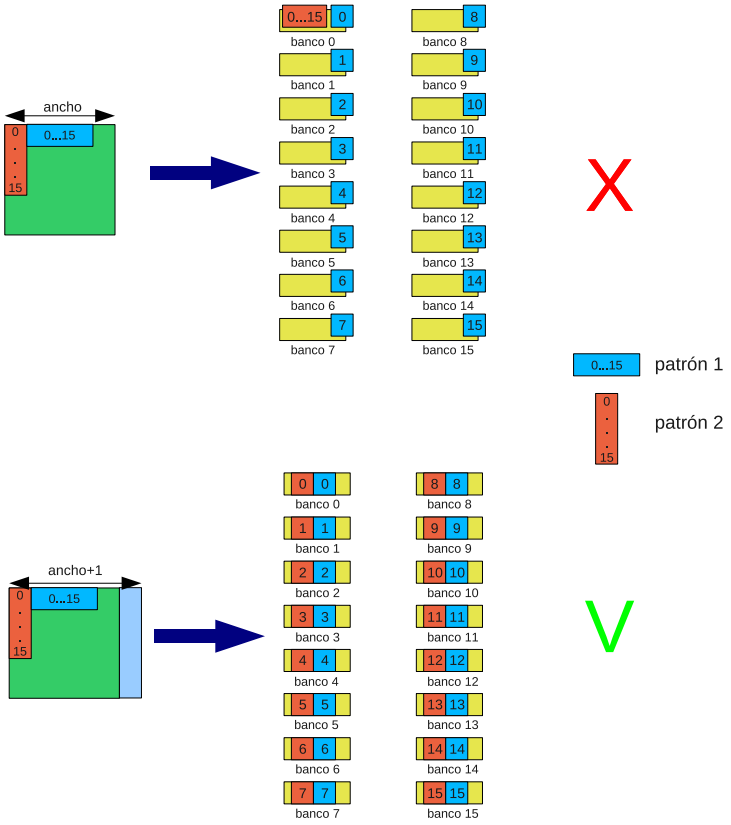


Figura 3.3: Padding para evitar conflictos en los bancos de memoria compartida

La Figura 3.3 muestra dos situaciones a la hora de alojar una matriz de dos dimensiones en memoria compartida. La primera aloja $alto * ancho$ elementos. En la segunda se realiza un *padding* de un elemento y se aloja $alto * (ancho + 1)$. Si $ancho$ es multiplo de 16, entonces alojar $alto * ancho$ elementos provoca que todos los elementos de las mismas columnas se alojen en el mismo banco de memoria. Ahora bien, tenemos dos patrones de acceso: en el primero los warps acceden a elementos consecutivos de una fila; mientras que en el segundo patrón los warps acceden a elementos consecutivos de las columnas. En la primera situación, el patrón de acceso por filas no provoca conflicto; sin embargo, el patrón de acceso por columnas provoca que todos los hilos accedan al mismo banco, provocando 16 conflictos en cada acceso. Al alojar $ancho + 1$ evitamos el problema ya que ahora los elementos consecutivos verticalmente no se alojan en el mismo banco de la memoria compartida. De esta forma, hacer un *padding* de 1 evita los conflictos de memoria respectando el patrón de acceso.

3.1.2. Accesos unificados (Coalesced)

En el apartado anterior vimos cómo minimizar los accesos al espacio de memoria global. Aunque la memoria compartida puede reducir enormemente el número de accesos a la memoria global, hay veces que se debe seguir accediendo a memoria global (aunque sea para leer los datos que se llevarán a la memoria compartida). Como se explicó en la Sección 2.2.1.2, los accesos a la memoria global se hacen a nivel de medio warp y una petición a memoria global puede provocar desde una hasta dieciséis transacciones con la memoria global. Es decir, cuando un warp ejecuta una instrucción a memoria global genera de dos a treinta y dos transacciones en función del número de segmentos distintos donde están los datos. A menor número de transacciones, más unificado (*coalesced*) es el acceso y se aprovecha mejor el ancho de banda a memoria. Esto implica una mejora en el rendimiento. Para realizar los accesos coalesced, hay que tener en cuenta que la tarjeta es capaz de leer de la memoria global palabras de 4, 8 y 16 bytes a registros en una única instrucción. Si el tamaño del tipo de datos a leer es mayor de 16 bytes, entonces se generan varias instrucciones de lectura. Como el mecanismo de unificación une en una o más transacciones una instrucción, si el tamaño de los datos es mayor de 16 bytes no se puede sacar provecho de los accesos coalesced (ver Figura 3.4).

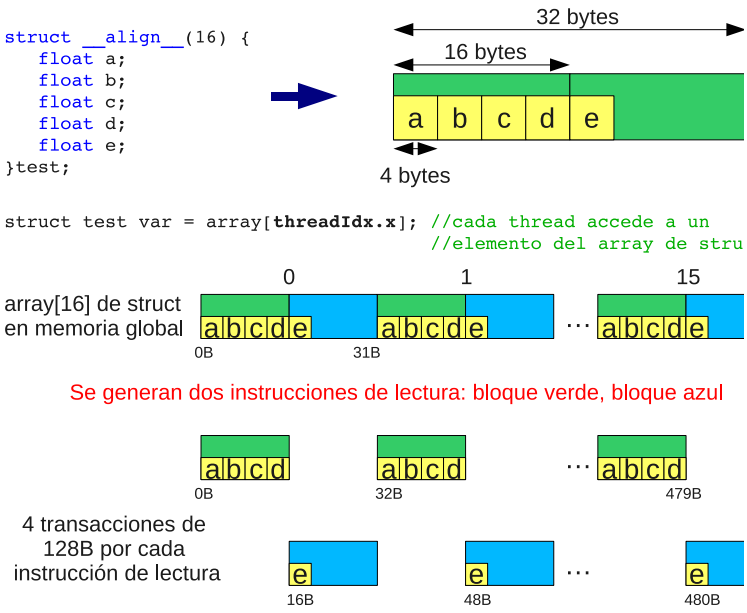


Figura 3.4: Acceso a datos mayores de 16 bytes

En lugar de ello, es preferible, como muestra la Figura 3.5, partir la estructura en varias estructuras de a lo sumo 16 bytes para unificar accesos.

Como se vió en la Sección 2.2.1.2, los patrones de acceso únicamente tienen que respetar que se acceda al mismo segmento de memoria. Por ello, es habitual utilizar el ID de cada hilo ($DIR = BASEDIR + tid$) para acceder a los datos que debe procesar. Así, el que un patrón genere accesos coalesced depende prácticamente de la geometría de los bloques. Como la memoria es lineal, hay que tener en cuenta que al manejar matrices de varias dimensiones, éstas se mapean de forma secuencial. Basando el patrón de acceso en el ID de cada hilo ($DIR = BASEDIR + ty * WIDTH + tx$), han de cumplirse al menos dos condiciones para garantizar que los accesos son coalesced:

- La geometría del bloque debe ser tal que el ancho del bloque sea multiplo del tamaño de medio warp (16).
- **width** sea multiplo de 16.

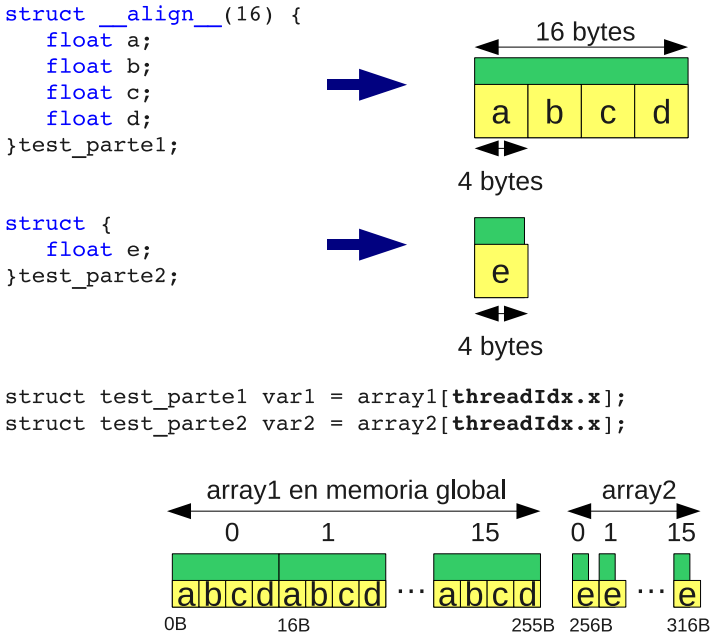


Figura 3.5: Acceso a datos estructurados

En particular, esto quiere decir que cualquier matriz cuyo ancho no sea multiplo de 16 será accedida de forma más lenta. Para evitar esto, se puede seguir una estrategia de *padding* similar a la que se usó con la memoria compartida. En este sentido, CUDA ofrece directamente la solución. En lugar de reservar memoria con la función **cudaMalloc()**, se puede usar **cudaMallocPitch()** y **cudaMalloc3D()** para matrices de 2D y 3D respectivamente. Estas funciones añaden memoria extra para cumplir con las restricciones de acceso.

3.2. Maximizar Occupancy

Dejando de lado el sistema de memoria, ahora vamos a centrarnos en el grado de paralelismo. Para ello vamos a estudiar cúantos de los recursos de los multiprocesadores están en uso. El concepto de *Occupancy* refleja la cantidad de recursos en uso de un multiprocesador. Se define como la relación entre los warps activos y el máximo de warps activos de un multiprocesador. Entonces, a mayor *occupancy* mayor uso de los multiprocesadores. Por lo tanto, una buena estrategia de optimización debería ser tratar de elevar el *occupancy* al máximo.

Elevar el *occupancy* es sinónimo de elevar la *capacidad de computación* de los multiprocesadores, y es una buena estrategia para elevar el paralelismo. Teniendo en cuenta que los warps se componen de 32 hilos y que un multiprocesador puede ejecutar 1024 hilos, el máximo número de warps es 32. Sin embargo no siempre es posible conseguir ejecutar 32 warps a la vez. Los siguientes factores limitan el número de warps activos:

- El número de hilos por bloque y el número de bloques de hilos asignados al multiprocesador. Los multiprocesadores puede ejecutar a lo sumo ocho bloques de hilos en paralelo. Si el número de hilos por bloque es menor de 128 (4 warps) nunca se llegará al máximo occupancy. Por otro lado, un bloque no se puede definir con más de 512 hilos (16 warps), por lo que al menos ha de haber dos bloques activos en el multiprocesador para obtener el máximo occupancy.

- El número de registros asignados a cada hilo. Cada multiprocesador cuenta con 16384 registros que asigna en bloques de 512 a los bloques de hilos. Esto hace que, habiendo hilos suficientes, cada hilo deba usar a lo sumo 16 registros para obtener el occupancy máximo.

- La cantidad de memoria compartida asignada a cada bloque de hilos. Cada multi-

procesador cuenta con 16KB de de memoria compartida. La cantidad de memoria compartida que usa cada bloque de hilos queda definida por el desarrollador a la hora de programar un kernel. Como ajustar el uso de la memoria compartida es otra estrategia de optimización que se vio en la Sección 3.1.1.

El Occupancy se puede calcular de forma estática según las siguientes ecuaciones. Los datos a conocer son el número de hilos por bloque (**#threads**), número de registros por hilo (**#regs**) y cantidad de memoria compartida (**#shMem**) por cada bloque de hilos.

$$Occupancy = warps\_activos\_por\_SM/32 \tag{3.2}$$

$$\#warps = \textbf{\#threads}/32 \tag{3.3}$$

$$warps\_activos\_por\_SM = bloques\_activos\_por\_SM * \#warps \tag{3.4}$$

$$bloques\_activos\_por\_SM = min(limite\_por\_warps, limite\_por\_regs, limite\_por\_shared) \tag{3.5}$$

$$registros\_por\_bloque = multiploSuperior(multiploSuperior(\#warps, 2)*\textbf{\#regs}*32, 512) \tag{3.6}$$

$$shared\_por\_bloque = multiploInferior(\textbf{\#shMem}, 512) \tag{3.7}$$

$$limite\_por\_warps = min(8, 32/\#warps) \tag{3.8}$$

$$limite\_por\_regs = multiploInferior(total\_registros/registros\_por\_bloque) \tag{3.9}$$

$$limite\_por\_shared = multiploInferior(total\_shared/shared\_por\_bloque) \tag{3.10}$$

Como veremos en la Sección 3.4, no es necesario calcular el occupancy siguiendo estas ecuaciones, pero dan una idea de cómo se ocupan los recursos.

Por tanto, si nuestro objetivo es maximizar el occupancy, tenemos que incrementar el número de warps, ya sea aumentando el número de hilos por bloque, disminuyendo el número de registros o disminuyendo la cantidad de memoria compartida. En el primer caso, conseguiremos aumentar el occupancy a base de ejecutar más hilos y menos bloques; y en los dos siguientes, aumentamos el occupancy a base de ejecutar más bloques en paralelo.

Las cuestiones de geometría de bloques se discuten en la siguiente Sección, mientras que ya se habló de la memoria compartida en la Sección 3.1.1. Únicamente nos queda discutir acerca del número de registros. En realidad, no existe un control directo sobre el uso de los registros de los multiprocesadores, al menos no a alto nivel. Todas las variables

(excepto los arrays) definidas en un kernel se alojan en registros, sin embargo el cálculo de los registros totales que se necesitan los realiza el compilador aplicando sus propias estrategias de de planificación de registros. Para saber cúantos registros utiliza un kernel, se debe compilar con la opción **–ptxas-options=-v**. Al compilar con esta opción se indica en la salida de la compilación el número de registros que utiliza cada hilo del kernel compilado. Cualquier valor igual o menor de 16 es perfecto, ya que no impone ninguna restricción al occupancy. Sin embargo, valores más altos afectan al occupancy en mayor o menor medida dependiendo del número de hilos por bloque. Como las ejecuciones se realizan por bloques de hilos, aunque todos los hilos menos uno tengan registros suficientes para ejecutarse el bloque entero debe esperar. Por ello, bloques grandes (de 256 ó 512 hilos) que se vean limitados por el número de registros, merman el occupancy a razón de 25 % y 50 % respectivamente.

Por otro lado, es posible forzar el número máximo de registros tratando de limitarlos en tiempo de compilación. La opción de compilación **–maxrregcount=X** limita el número de registros de los kernels compilados al valor X. Esta estrategia tiene contrapartidas, pues hace uso de memoria local[1] para hacer *spilling* cuando llega al límite de registros. El spilling puede llegar a penalizar gravemente el rendimiento, aunque reducir el número de registros en 2 ó 3 por hilo no genera un tráfico de spilling crítico.

3.2.1. Geometría de bloques

Como se ha visto en las secciones anteriores, la geometría de los bloques está íntimamente ligada con optimizaciones de memoria y paralelismo. Si bien es cierto que no hay una geometría óptima general, sí que existen algunas pautas para encontrarla. En primer lugar define los patrones de acceso a la memoria. Según hemos visto en la Sección 3.1.2, si los datos se organizan en matrices de varias dimensiones, el ancho del bloque debe ser multiplo de 16. Además, para conseguir un occupancy alto, cada bloque de hilos debe tener entre 192 y 256 hilos; incluso 512 en algunos casos. Esto convierte a los bloques de tamaño 16x16, 16x32 y 32x16 en los bloques que generalmente obtienen mejores resultados.

Por otro lado, también es importante discutir sobre cúanto trabajo se asigna a cada hilo. Normalmente, el bloque de datos que procesa un bloque de hilos está subordinado al bloque de memoria compartida que se asigna al bloque de hilos. Es habitual que, por límites de memoria compartida, esa relación sea 1:1. Pero en casos en los que no se usa memoria compartida, o se usa una cantidad pequeña, esa relación puede cambiar de forma

[1] recúerdese que está situada en el espacio de memoria global

que un hilo procese varios datos. De esta forma se realiza un *tiling* a nivel de memoria compartida o de cómputo. En realidad, siempre que un hilo pueda computar más de un dato de forma *gratuita* debe hacerlo. Por gratuita se entiende que el hilo únicamente ejecutará operaciones y nunca accesos a memoria.

El hecho de que un bloque de hilos procese más de un dato por hilo tiene ciertas ventajas. En primer lugar evita redundancias interbloque en las memorias compartidas lo que ayuda a que en conjunto, el uso de la memoria compartida sea más eficiente. Puede disminuir el tráfico con la memoria global. Como contrapartida, disminuye el número total de bloques de hilos. Esto puede hacer que las latencias a memoria no se oculten lo suficiente en caso de tener muy pocos bloques. Por todo ello, esta solución suele encajar cuando las dimensiones del problema son lo suficientemente grandes como para generar un número de bloques por multiprocesador adecuado.

3.3. Flujo de instrucciones

Las instrucciones de control de flujo tales como saltos tienen también un gran impacto en el rendimiento. Si la condición de salto provoca que hilos del mismo warp tomen caminos distintos, la ejecución deja de ser paralela y deben serializarse ambos caminos del salto. Primero, los hilos que toman el salto ejecutan una rama y depués los hilos restantes ejecutan la otra rama. Por supuesto, esto repercute notablemente en el rendimiento, no sólo por el aumento de instrucciones a ejecutar, sino porque los hilos dejan de ejecutarse en paralelo. Este efecto se ilustra en la Figura 3.6.

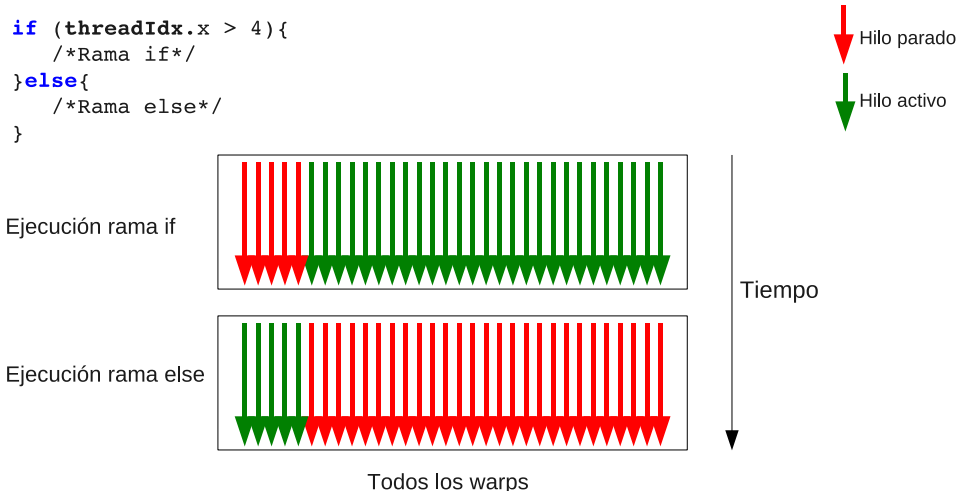


Figura 3.6: Ejecución en serie de los saltos

Para evitar esto, se debe aumentar la granularidad de los saltos a nivel de warp o un múltiplo. De forma que todos los hilos de un mismo warp sigan el mismo camino y la ejecución continue siendo paralela.

```
if(threadIdx.x > 4) //provoca que el warp serialice la ejecucion
  ...
else
  ...


if(threadIdx.x/WARP_SIZE > 4) //todos los hilos del mismo warp
  ...                         //siguen el mismo camino y no se
else                          //serializa la ejecucion
  ...
```

Figura 3.7: Evitar divergencia en los warps

Por último, hablaremos de mejoras a nivel de instrucción. A lo largo de todo el capítulo se ha hablado de lo costosas que son las operaciones en memoria y de tratar de ocultarlas a base de aumentar el paralelismo. Otro método para ocultarlas es aumentar el paralelismo a nivel de instrucción (ILP). El unrolling de bucles es un técnica con la que es posible aumentar el ILP y reducir el número de instrucciones dinámicas a costa de aumentar el tamaño del código estático y a costa de hacer un mayor uso de registros. Por defecto el

compilador trata de desenrollar todos bucles. Sin embargo, en la práctica, sólo es capaz de desenrollar aquellos bucles que en tiempo de compilación tiene un número de iteraciones fijo, lo cual no es habitual. El desenrollado no solo aumenta la relación entre el número de instrucciones de cómputo frente memoria, sino que también elimina las instrucciones de control, evitando la ejecución de instrucciones que no ayudan al progreso del cómputo.

```
for (int k=0; k<BLOCK_SIZE; ++k)
    Pvalue += Ms[ty][k] * Ns[k][tx];
```

a) El bucle provoca exceso de instrucciones

```
Pvalue += Ms[ty][k] * Ns[k][tx] + ...
                  Ms[ty][k+15]*Ns[k+15][tx];
```

b) El unrolling elimina ese exceso

Figura 3.8: El unrolling mejora la ejecución de instrucciones

Por ejemplo, el bucle de la Figura 3.8(a) ejecuta 2 instrucciones aritméticas en punto flotante, una instrucción de salto, dos instrucciones aritméticas de direccionamiento y una instruccion para incrementar el contador. Entonces sólo 1/3 de las instrucciones ejecutadas son de computo del algoritmo. Esta mezcla de instrucciones limita el ancho de banda de procesado de instrucciones, de forma que no se consiga más de 1/3 del potencial.

Sin embargo, en la Figura 3.8(b) se expresa el mismo cómputo completamente desenrollado. Se consigue eliminar la instrucción de salto y la actualización del contador. Además como los direccionamientos son constantes, el compilador puede optimizar el código utilizando direccionamiento basado en desplazamientos para eliminar las instrucciones de direccionamiento. Como resultado, se aumenta el rendimiento. Aunque es conveniente realizar el desenrrollado de forma manual para saber exactamente las instrucciones que se ejecutan, no es necesario. Es posible utilizar sentencias de preprocesado antes de los bucles para que la labor de desenrrollado recaiga en el compilador.

3.4. Profiler CUDA

Hasta ahora hemos visto técnicas generales para optimizar los kernels en la GPU a alto nivel. Ahora necesitamos algún método para clasificar y evaluar dichas optimizaciones, es

decir, cuantificar de alguna forma por qué unas son mejores que otras. Para ello, vamos a recurrir a técnicas de *profiling*. Consiste en obtener información dinámica de un algoritmo durante su ejecución. Para ello, la GPU dispone de una serie de contadores hardware para contabilizar ciertos aspectos de bajo nivel de los kernels ejecutados. Esta será nuestra herramienta para tomar valores de las distintas métricas. Además al contar instrucciones máquina, trataremos de relacionar los cambios a alto nivel con las variaciones en los resultados de los contadores.

No todos los multiprocesadores de la GPU cuentan con estos contadores. Únicamente el primer TPC cuenta con ellos, y aún así, algunos contadores solo están disponibles para el primer multiprocesador del TPC. Como consecuencia, si se quiere que las medidas sean correctas, es necesario lanzar una gran cantidad de bloques de hilos. De este modo, los bloques se distribuirán por todos los TPC de forma uniforme. Los valores de los contadores no corresponden a la actividad individual de cada hilo. En lugar de eso, los contadores cuentan eventos de warps enteros.

Además de los contadores, el profiler CUDA proporciona información acerca del tiempo de ejecución, occupancy, uso de registros y uso de memoria compartida. Las tablas Cuadro 3.3 y Cuadro 3.4 muestran los contadores disponibles en la GPU, en la primera se muestran los contadores disponibles en el TPC cero. Los valores de estos contadores son acumulativos para todos los bloques de hilos que se ejecutan en el primer TPC. Nótese que hay tres multiprocesadores por TPC. Los contadores relacionados con la memoria global cuentan el número de accesos o transacciones a memoria y no son por warp; la segunda tabla muestra los contadores disponibles en el multiprocesador cero del TPC cero. Al igual que los anteriores, los valores de estos contadores son acumulativos para todos los bloques de hilos que se ejecutan en el primer multiprocesador. Estos contadores se incrementan en uno por cada warp.

gld_coherent	Número de lecturas coalesced de memoria global
gld_request	Número de peticiones de lectura a memoria global
gld_32/64/128b	Número de transacciones de lectura con memoria global de 32, 64 y 128 bytes respectivamente
gst_coherent	Número de escrituras coalesced de memoria global
gst_request	Número de peticiones de escritura a memoria global
gst_32/64/128b	Número de transacciones de escritura con memoria global de 32, 64 y 128 bytes respectivamente
local_load	Número de lecturas de memoria local
local_store	Número de escrituras en memoria local
cta_launched	Número de bloques de hilos lanzados en el TPC
tlb_hit	Número de aciertos en la caché de instrucciones y caché de constantes
tlb_miss	Número de fallos en la caché de instrucciones y caché de constantes

Cuadro 3.3: Contadores disponibles en el TPC cero

branch	Número de saltos tomados por los hilos en la ejecución del kernel. Este contador se incrementa en uno si al menos un hilo del warp toma el salto
divergent_branch	Número de saltos divergentes en el warp. Este contador se incrementa en uno si al menos un hilo diverge en un salto
instructions	Número de instrucciones ejecutadas
warp_serialize	Número de warps que serializan el acceso a memoria compartida o constante debido a conflictos en los bancos
sm_cta_launched	Número de bloques de hilos lanzados en el multiprocesador

Cuadro 3.4: Contadores disponibles en el multiprocesador cero del TPC cero

Las lecturas de estos todos estos contadores son las que usaremos como métricas para modelar las aplicaciones. Podemos analizar en datalle el impacto de las transformaciones a alto nivel en todos los factores de rendimiento presentados en este capítulo.

Capítulo 4

Aplicaciones de estudio

A lo largo del capítulo anterior hemos visto de forma general algunas técnicas de optimización para códigos CUDA. En este capítulo aplicaremos esas técnicas a varios algoritmos de uso cotidiano. Dada la naturaleza vectorial de las tarjetas gráficas, es lógico que traten de usarse en algoritmos de cálculo vectorial y matricial. La *multiplicación de matrices* constituye uno de los algoritmos básicos en el desarrollo de algoritmos matriciales, ya que aunque no es demasiado complejo, se utiliza como operación básica en otros algoritmos matriciales más grandes.

Como el uso de los recursos en GPU dependen mucho del algoritmo a implementar, hemos querido estudiar en detalle otro algoritmo además de la multiplicación de matrices. Aunque la complejidad algorítmica de las convoluciones es incluso menor que la multiplicación de matrices, el mundo multimedia las ha convertido en partes habituales de algoritmos más grandes en el tratamiento de imagenes, video y audio. Muchas veces estos algoritmos requieren de la característica tiempo real, y por ello nos parece un algoritmo a tener en cuenta para su estudio en la GPU.

Por otro lado, para comprobar el impacto de las implementaciones de estos algoritmos , hemos implementado un algoritmo matricial mayor tanto en CPU como en GPU para comparar su rendimiento. El algoritmo implementado es una versión supervisada del NMF (*Non negative Matrix Factorization*) que puede encontrarse en el Apéndice A.

4.1. Multiplicación matrices

Como algoritmo, la multiplicación de matrices resulta una operación costosa para realizar en la CPU, sobretodo teniendo en cuenta que forma parte de algoritmos más grandes. Supone muchos accesos a memoria (la mayoría duplicados). Además es un algoritmo muy uniforme en el sentido que siempre realiza el mismo cómputo sobre distintos datos. Estos motivos hacen que se ajuste al modelo SIMT de la GPU. A continuación se muestra la implementación clásica del algoritmo en la CPU. Partiremos de ella como versión inicial para la GPU.

```
for(int i=0; i<N; ++i){
    for(int j=0; j<M; ++j){
        for(int k=0; k<L; ++k){
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Figura 4.1: Implementación clásica de la multiplicación de matrices

En primer lugar debemos concretar qué hará cada hilo. Esta decisión simplemente marca el orden de los bucles de la Figura 4.1, y cambia tanto los patrones de acceso a memoria como el número de accesos totales:

- Cada hilo produce un resultado de C.
- Maximizar el reuso de A, cada hilo produce un resultado parcial de una fila (o fracción) de C.
- Maximizar el reuso de B, cada hilo produce un resultado parcial de una columna (o fracción) de C.

Las opciones de maximizar A ó B producen resultados parciales que, implementado en la GPU, se traducen en escrituras en memoria global intermedias ya que no hay memoria compartida suficiente para albergar las matrices. Como hemos visto, las operaciones en memoria global son muy costosas, por ello, estas son malas implementaciones para la GPU. Así optamos por la primera opción, cada hilo producirá un resultado de C tal y como se muestra en la Figura 4.2.

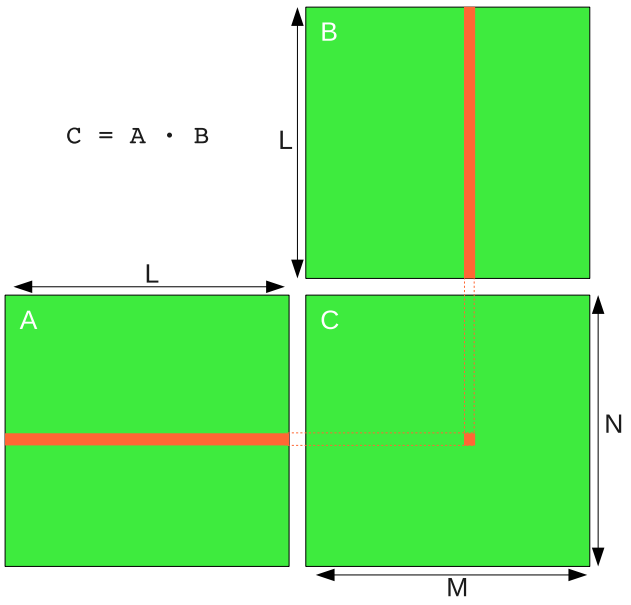


Figura 4.2: Multiplicación de matrices - tarea asignada a cada hilo

Cada hilo accede L veces a la matriz A (una fila entera), L veces a la matriz B (una columna entera) y escribe una vez en la matriz C, es decir $2*L$ lecturas de memoria global y una escritura en memoria global por hilo. En cuanto a la configuración, ha de haber un hilo por elemento a calcular. Esto es $N*M$ hilos repartidos entre los bloques. El kernel que implementa esta versión se muestra a continuación:

```
__global__ void basic_matrix_mult(float *A, float *B, float *C,
                                  int N, int M, int L){
    int tx = blockDim.x * blockIdx.x + threadIdx.x;
    int ty = blockDim.y * blockIdx.y + threadIdx.y;
    float valor = 0.0f;

    for(int i=0; i<L; ++i){
        valor += A[ty * L + i] * B[i * M + tx];
    }
    C[ty * M + tx] = valor;
}
```

Figura 4.3: Versión básica de la multiplicación de matrices en GPU

Sin embargo, esta implementación, al igual que el algoritmo clásico en CPU, realiza muchos accesos a memoria innecesarios disminuyendo notablemente el rendimiento. Todos los hilos que procesan elementos de la misma fila de C leen la misma fila de A varias veces, y todos los hilos que procesan elementos de la misma columna de C leen la misma columna de B varias veces. La mayoría de accesos a la memoria global sean redundantes y merman el rendimiento total. Esto se debe a que los hilos no colaboran entre si, funcionan de forma independiente y no se saca provecho del sistema de memoria.

4.1.1. Optimizar el uso del sistema de memoria

El código de la Figura 4.3, aunque es más rápido que en la CPU, presenta ciertas carencias a la hora de aprovechar los recursos de la GPU. En primer lugar, redunda accesos a memoria global y en segundo todos los accesos a la matriz A son no coalesced.

Como primera optimización vamos a maximizar el reuso de datos, para ello aplicaremos un tiling sobre la memoria compartida. Con ello vamos a evitar los principales problemas de la implementación anterior. Al aplicar el tiling, el kernel se divide en dos partes perfectamente diferenciadas. Una primera de carga, en la que el bloque de hilos lleva a memoria compartida un trozo de A y otro trozo de B. Y una segunda en la que los hilos trabajan con los datos de la memoria compartida para obtener los resultados de un trozo de C.

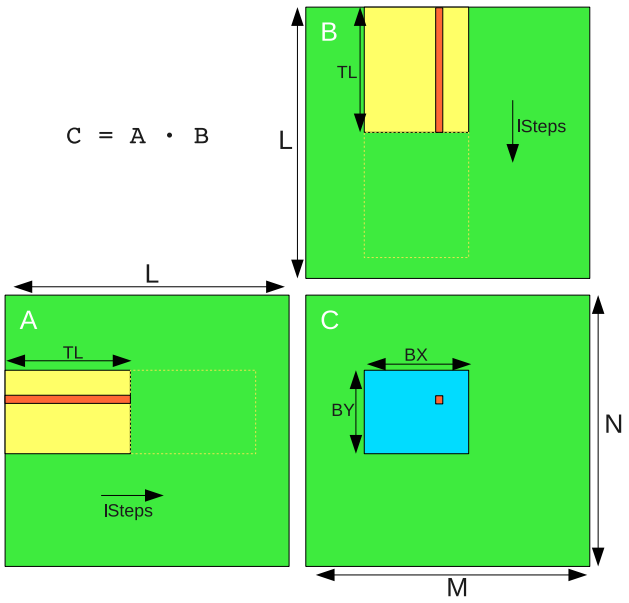


Figura 4.4: Multiplicación de matrices con memoria compartida

Como la memoria compartida es muy limitada en cuanto a tamaño, no se puede realizar todo el proceso con una única fase de lectura y cómputo. En lugar de ello, se define el parámetro TL y se itera a lo largo de L/TL pasos. En cada paso, los bloques llevan a memoria compartida $by * TL$ elementos de A y $TL * bx$ elementos de B (siendo bx y by el número de hilos del bloque en las dimensiones X e Y respectivamente). Esto reduce el número de accesos totales a memoria global, pues los datos de A y B se leen una única vez de memoria global. El patrón de acceso a la matriz A pasa a ser coalesced (siempre que bx sea multiplo de 16), ya que cada hilo lee posiciones consecutivas; y el acceso a la memoria compartida no provoca conflictos en el acceso. El valor del parámetro TL resulta decisivo en el rendimiento tal y como veremos en el Capítulo 5, pues controla la cantidad de memoria compartida que usa cada bloque:

$$memoria\_compartida\_por\_bloque = (by + bx) * TL \tag{4.1}$$

Además, TL puede provocar que algunos hilos estén ociosos en la fase de lectura. Si TL se define de forma que es menor que bx, entonces todos los hilos con ID mayor que TL no hacen nada en durante la lectura del tile de A. Lo mismo ocurre si TL es menor que by durante la lectura del tile de B.

Tanto el código mostrado en la Figura 4.3 y en la Figura 4.5 están incompletos ya que solo tratan de ilustrar los elementos clave del kernel. A estos códigos les falta toda la parte de control para que ningún hilo procese elementos fuera de las matrices A, B y C.

Aunque son condiciones a nivel de bloque, y por lo tanto no conllevan divergencia en los warps, si pueden llegar a suponer un gran aumento en las instrucciones dinámicas en caso de encontrarse dentro de bucles. En realidad no todos los hilos deberían realizar estas comprobaciones. Solo los hilos de los *bloques frontera* pueden acceder a posiciones fuera de la matriz C, por lo tanto deberían ser lo únicos en realizar comprobaciones de los límites. Por ello, el siguiente paso es modificar el código de forma que lo primero que hace el kernel es comprobar si es bloque frontera, en tal caso ejecuta el algoritmo con comprobaciones; y si no es bloque frontera ejecuta el algoritmo sin comprobaciones. Como los bloques se dividen en warps completos nunca habrá divergencia en este salto y solo se ejecutará uno de los dos caminos. En general, todas las instrucciones condicionales deben flotar en la jerarquía de bucles aunque esto suponga duplicar, en la mayoría de los casos, el código fuente.

A la vez que esta optimización en el orden de las instrucciones, también vamos a eliminar instrucciones redundantes a costa de ampliar el uso de registros. El principal objetivo de esta optimización son las instrucciones aritméticas para el cálculo de las direcciones de los arrays. Estas operaciones se repiten a causa de los bucles y pueden ser simplificadas utilizando variables (lo que se traduce en registros) para que almacenar los cálculos.

```
__global__ void shared_matrix_mult(float *A, float *B, float *C,
                                   int N, int M, int L){
   __shared__ float tileA[BY][TL];
   __shared__ float tileB[TL][BX];

   int tx = blockDim.x * blockIdx.x + threadIdx.x;
   int ty = blockDim.y * blockIdx.y + threadIdx.y;

   int lSteps = L/TL;

   int l_tile_x = TL/blockDim.x;
   int l_tile_y = TL/blockDim.y;

   float valor = 0.0f;

   for(int lSt=0; lSt<lSteps; ++lSt){

      //Lectura de los tiles a memoria compartida
      for(int u=0; u<l_tile_x; ++u){
         tileA[threadIdx.y][u * blockDim.x + threadIdx.x] =
               A[ty * L + (lSt * TL + u * blockDim.x + threadIdx.x)];
      }

      for(int u=0; u<l_tile_y; ++u){
         tileB[u * blockDim.y + threadIdx.y][threadIdx.x] =
               B[(lSt * TL + u * blockDim.y + threadIdx.y) * M + tx];
      }

      __syncthreads();

      //Operar con los tiles en memoria compartida
      for(int i=0; i<TL; ++i){
         valor += tileA[threadIdx.y][i] * tileB[i][threadIdx.x];
      }

      __syncthreads();
   }

   C[ty * M + tx] = valor;
}
```

Figura 4.5: Versión con memoria compartida de la multiplicación de matrices en GPU

Al aplicar el tiling sobre la memoria compartida se produce un efecto interbloque negativo. Bloques de hilos que procesen elementos de las mismas filas mantienen copias iguales de A en sus memorias compartidas. Lo mismo ocurre con los bloques de hilos que procesan elementos de las mismas columnas y la matriz B. Hay que recordar que hay muy poca memoria compartida en los multiprocesadores y no debe ser malgastada manteniendo copias de los mismos datos.

Para tratar de minimizar el efecto vamos a aumentar el area de elementos sobre los que actua un bloque. Cada bloque utilizará más memoria compartida pero también procesa más elementos de la matriz C. De esta forma conseguimos menos bloques activos debido a la memoria compartida y menos copias de interbloque. Para que el número de hilos no sea también una limitación en cuanto al número de elementos a calcular, cada hilo procesará varios elementos de C. Así obtenemos otro tiling a nivel de bloque. Qué elemento procese qué hilo no puede ser al azar, pues afecta a la forma de realizar los accesos a memoria. La forma más eficiente de hacerlo es utilizar el número de hilos en cada dirección como desplazamiento, de forma que cada hilo procesa elementos cada BX posiciones en horizontal y cada BY en vertical, tal y como muestra la Figura 4.6.

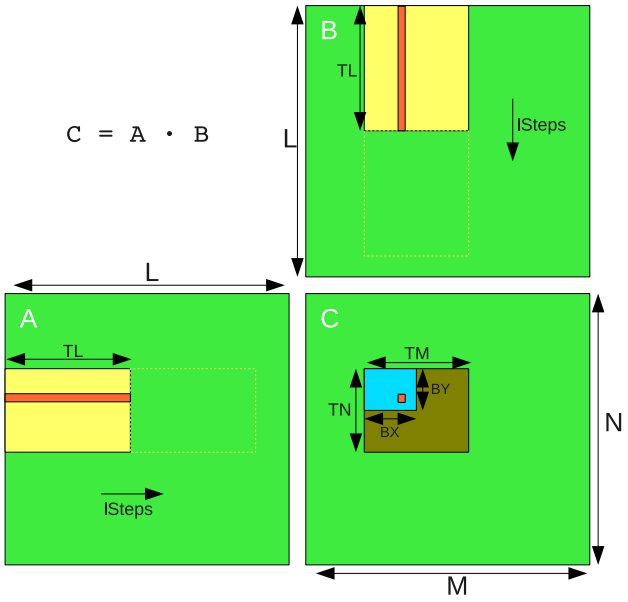


Figura 4.6: Multiplicación de matrices con memoria compartida y varios elementos por hilo

Con esta nueva implementación aparecen dos nuevos parámetros TN y TM, que de-

ben ser multiplos del número de hilos en cada dirección para obtener mejores resultados (de lo contrario debe añadirse código de control que involucra condiciones posiblemente divergentes). Es importante especificar que la parte de carga de elementos de memoria global a memoria compartida se hace en un único paso. Así el tamaño de los tiles de la matriz A y la matriz B que se llevan a memoria compartida pasan a ser de $TN * TL$ y de $TL*TM$ respectivamente. Además, cada hilo debe almacenar varios resultados intermedios donde antes solo tenía que almacenar uno. Aquí hemos realizado dos implementaciones: una genérica en la que cada hilo utiliza un array para los resultados intermedios. Y otra específica en la que cada hilo utiliza variables escalares para los resultados intermedios. Como ya explicamos anteriormente los arrays se mapean en memoria local, mientras que las variables escalares se mapean en los registros. De esta forma podemos ver el impacto de la memoria local.

Aún así, con esta implementación, el hecho de que cada hilo procese más de un elemento ($TM >= 2 * BX$ y/o $TN >= 2 * BY$) hace que el consumo de la memoria compartida sea muy grande y limite el tamaño del trozo de C que se está calculando, asi como el lanzamiento de hilos totales. Por ello, vamos a plantear otra estrategia que permite aumentar un poco más el tamaño de elementos a procesar. Esta nueva implementación realiza la lectura de datos desde memoria global a memoria compartida en dos fases. Trae un tile entero de una de las matrices A o B y la mitad de la otra, realiza los cálculos intermedios, trae la otra mitad del tile y finalmente termina el cálculo. Esto permite procesar el doble de elementos de C usando la misma memoria compartida, además reduce, frente a otras implementaciones, las transacciones de memoria global a memoria compartida.

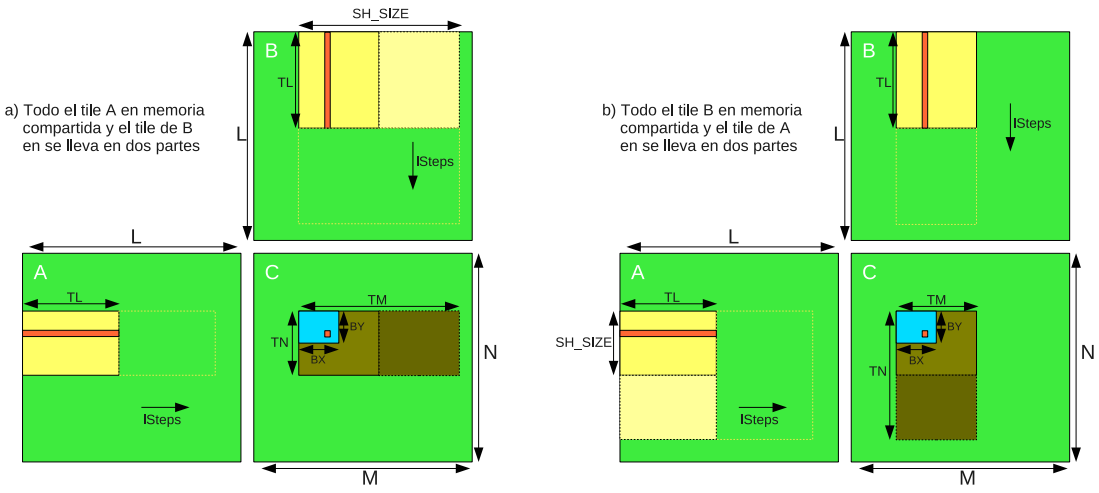


Figura 4.7: Multiplicación de matrices con dos fases de cargas en memoria compartida

4.2. Convoluciones

Las convoluciones se utilizan en muchas aplicaciones de ingeniería y matemáticas. En particular, muchos tipos de filtros de distorsión o detección de bordes en imágenes usan convoluciones. Matemáticamente, una convolución mide el índice de coincidencia entre dos funciones.

$$r(i) = (s * k)(i) = \int s(i-n)k(n)dn \tag{4.2}$$

En términos discretos se puede expresar como:

$$r(i) = (s * k)(i) = \sum_{n} s(i-n)k(n) \tag{4.3}$$

La convolución puede extenderse a dos dimensiones simplemente añadiendo los índices de la segunda dimensión:

$$r(i) = (s * k)(i,j) = \sum_{n}\sum_{m} s(i-n, j-m)k(n,m) \tag{4.4}$$

En el contexto del procesado de imagen, un filtro de convolución no es más que el producto escalar entre una máscara de convolución y los píxeles de la imagen de entrada que rodean cada pixel de salida. Este producto escalar es una operación paralela que se ajusta muy bien al hardware de computación altamente paralelo como es la GPU.

Generalmente, la convolución de dos dimensiones necesita $n * m$ multiplicaciones por cada pixel de salida, donde n y m son el ancho y el alto de la máscara de convolución. La *convolución separable* puede dividirse en dos operaciones de convolución consecutivas de una dimensión, y así solo realizar $n + m$ multiplicaciones por cada pixel de salida.

La implementación más simple de la convolución en CUDA es que cada bloque de hilos cargue un bloque de la imagen en memoria compartida, cada hilo realice la multiplicación punto a punto con la máscara de convolución, y finalmente escriba la suma en la imagen de salida situada en memoria global. La Figura 4.8 muestra este proceso.

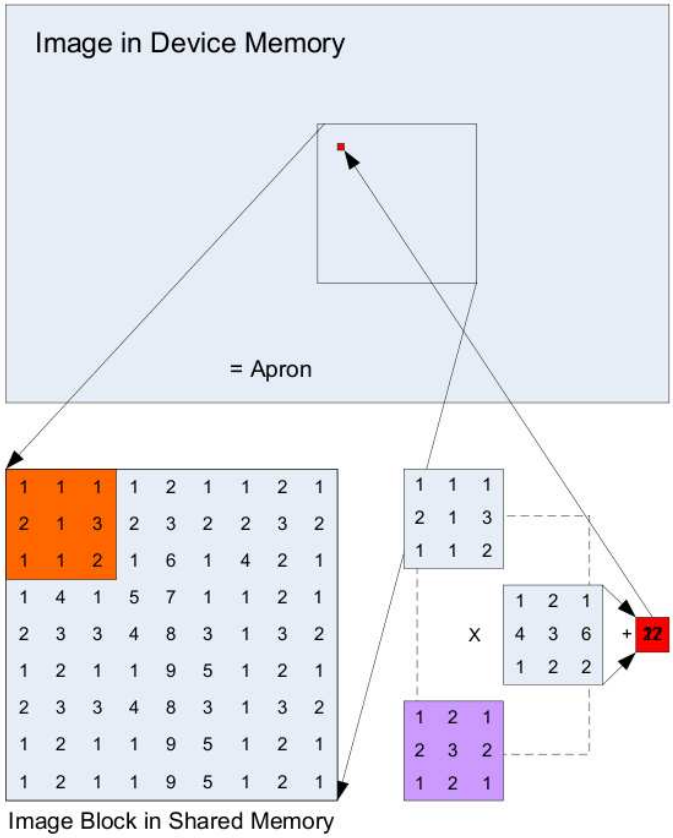


Figura 4.8: Implementación simple de la convolución. Un bloque de pixeles se carga en la memoria compartida. Para procesar un pixel de salida (rojo), se multiplica punto a punto una región de la imagen de entrada (naranja) con la máscara de convolución (morado), se suma el resultado y se escribe de nuevo en la imagen.

Para un tamaño razonable de la máscara de convolución, los píxeles de los bordes de la memoria compartida dependerán de otros píxeles que no están en la memoria compartida.

Existe un relleno alrededor del bloque de imagen con un ancho igual al radio de la máscara de convolución. El bloque de hilos también debe llevar a la memoria compartida este relleno para realizar el cómputo. La Figura 4.9 muestra como queda el algoritmo teniendo en cuenta este relleno.

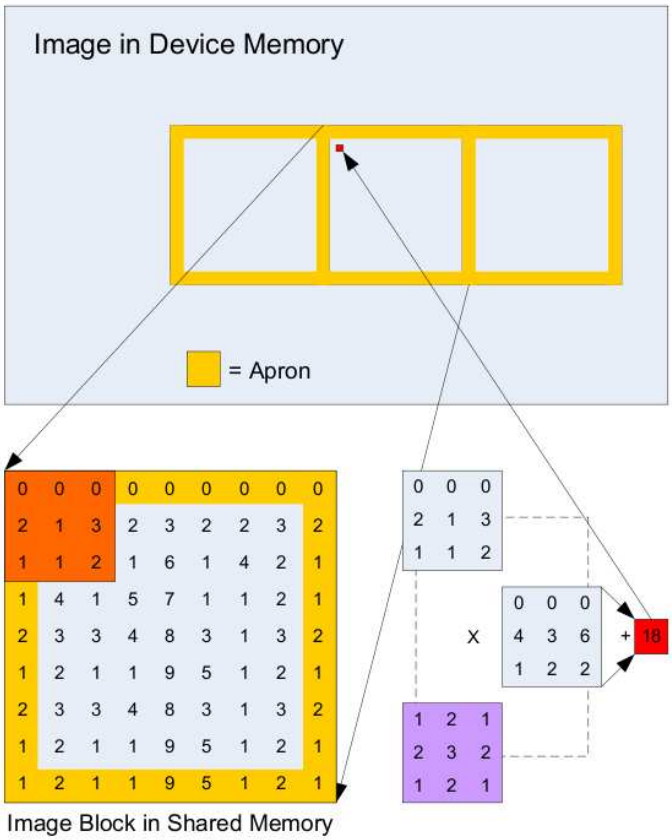


Figura 4.9: Convolución teniendo en cuenta los píxeles de relleno en memoria compartida

Aún falta por decidir donde está alojada la máscara de convolución y la geometría de los bloques. La máscara no es más que una matriz a la cual acceden los hilos, con la peculiaridad de que todos los hilos de un bloque acceden siempre al mismo elemento de la máscara. Debido a este patrón de acceso, alojar la máscara en la memoria compartida provocará conflictos en todos los accesos, además de obligar a que cada bloque de hilos mantenga una réplica de la máscara. Así, la solución más eficiente es alojar la máscara en la memoria de constantes que está optimizada para este tipo de patrón.

4.2.1. Optimizaciones

Si cada hilo carga un pixel en la memoria compartida, entonces todos los hilos que cargan los píxeles del relleno estarán ociosos durante la fase de cómputo. A medida que el radio de la máscara aumenta, el porcentaje de hilos ociosos aumenta también. Esto desperdicia gran parte del paralelismo, y con la cantidad limitada de memoria compartida, esta pérdida de paralelismo puede ser muy elevada con radios de máscara grandes.

Como ejemplo llevado al extremo, vamos a considerar un bloque de imagen de 16x16 y un radio de máscara también de 16. Esta configuración solo permite un bloque de hilos activo por multiprocesador. Asumiendo 4 bytes por pixel, un bloque usará 9216 bytes. Esto es más de la mitad de los 16KB de memoria compartida disponibles por multiprocesador. En este caso, solo 1/9 de los hilos estarán trabajando después de la etapa de carga.

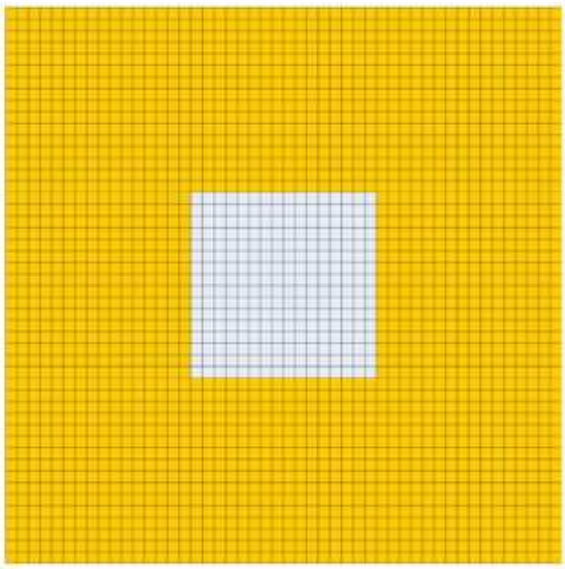


Figura 4.10: Si el radio de la máscara es grande en comparación al bloque de imagen, habrá muchos hilos ociosos durante la fase de cómputo

Podemos reducir el número de hilos ociosos reduciendo el número de hilos por bloque, y que durante la fase de carga, cada hilo cargue en memoria compartida más de un elemento. Así, podemos usar un hilo por cada elemento a computar. En este caso, para el ejemplo anterior, se divide la matriz en 9 cuadrados de 16x16, de forma que cada hilo lea 9 elementos. Si el relleno no es tan ancho como el bloque de hilos, entonces algunos hilos estarán ociosos en las iteraciones primera y última de la fase de carga.

Estos cambios no tienen por qué mejorar el rendimiento en todos los casos, ya que el aumenta de la complejidad del código puede no compensar el aumento en rendimiento. A continuación se explica la implementación de la convolución en dos pasadas. Como ya explicamos anteriormente, la implementación en dos pasadas ya reduce considerablemente la complejidad aritmética.

4.2.1.1. Convoluciones separable

Separar la operación de convolución en dos pasadas no solo beneficia a el hecho de reducir la complejidad aritmética o evitar que algunos hilos esten ociosos, también reduce el número de lecturas innecesarias. En el ejemplo anterior cada pixel del relleno se carga 9 veces debido al solapamiento entre bloques vecinos.

Si separamos el cómputo en dos pasadas: una horizontal (procesando sobre las filas) y otra vertical (procesando sobre las columnas), con una escritura en memoria global entre ambas, a lo sumo cada pixel se carga séis veces. Con bloques de imagen pequeños (16x16) esto no supone ningún beneficio. La principal ventaja se debe a que ya no es necesario cargar en memoria compartida la parte de arriba y abajo del relleno (en la pasada horizontal). Esto permite que se carguen más píxeles para procesar en cada bloque de hilos, es decir, ahora estamos limitados por el tamaño del bloque de hilos en lugar de por el tamaño de la memoria compartida. Para aumentar la eficiencia, cada hilo procesará más de un pixel. La forma de asignar qué píxeles procesa cada hilo será la misma que utilizábamos en la multiplicación de matrices: cada hilo procesa un pixel cada *ancho de bloque* píxeles.

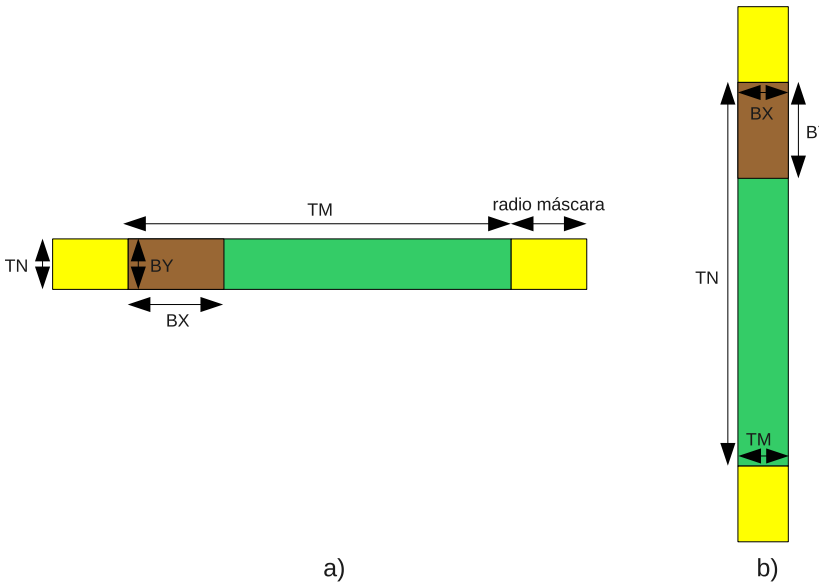


Figura 4.11: Convolución separable en dos pasadas: a) pasada horizontal (filas), b) pasada vertical (columnas)

Según la Figura 4.11, el kernel que procesa las filas carga en memoria compartida $(TM + 2 * MASK\_RAD) * BY$ elementos y $BX * (TN + 2 * MASK\_RAD)$ el kernel que procesa las columnas. A la hora de realizar la carga del relleno (bloques amarillo en la Figura 4.11) vamos a seguir dos estrategias distintas. La primera estrategia asume que el radio de la máscara se puede empaquetar en bloques del mismo tamaño que el bloque de hilos. De forma que si $MASK\_RAD\,\%BX! = 0$ (en la pasada horizontal, $MASK\_RAD\,\%BY! = 0$ en la vertical) habrá hilos ociosos en la fase de carga. Esta solución simplifica la etapa de carga frente a la segunda opción a base de sacrificar memoria compartida. Además, garantiza que los accesos a memoria global que realiza el kernel horizontal, sean coalesced en todas las arquitecturas de GPU. La segunda estrategia utiliza solo la cantidad exacta de memoria compartida para el relleno. Esta solución aumenta la complejidad del código y provoca divergencias en los warp salvo cuando el radio de la máscara coincide con ancho/alto del bloque de hilos.

Las condiciones de coalescing para el kernel vertical se cumplen siempre que el ancho del bloque de hilos sea multiplo de 16. En cuanto a la etapa de cómputo, cada hilo itera sobre los elementos de la máscara y sobre su región de memoria compartida. Realiza la multiplicación punto a punto y guarda la suma en memoria global de nuevo. Como

todos los hilos acceden simultaneamente a las mismas posiciones de la máscara, no hay conflictos de acceso (recuérdese que la máscara se aloja en memoria constante). Tampoco hay conflicto al acceder a la memoria compartida en ninguno de los kernels, ya que hilos del mismo *medio-warp* acceden a posiciones consecutivas de forma simultanea.

Capítulo 5

Resultados y análisis

Este capítulo muestra los resultados obtenidos de las implementaciones mostradas en el Capítulo 4. Se centra principalmente en analizar en profundidad los resultados de la multiplicación de matrices. También se analizan los resultados de la convolución y se muestra una comparativa de tiempos del algoritmo NMF (Ver Apéndice A). Todas las pruebas se han realizado sobre tarjetas gráficas modelo *Tesla C1060* y con el driver 3.1 de CUDA.

Numero de TPCs	10
Número de multiprocesadores	30
Número de total de procesadores	240
Memoria global	4GB
Memoria compartida por multiprocesador	16KB
Memoria constante por multiprocesador	16KB
Número de registros por multiprocesador	16384
Tamaño de warp	32
Frecuencia de reloj	1.3 GHz

Cuadro 5.1: Características de la tarjeta gráfica Tesla C1060

En el Capítulo 4 hemos visto varias implementaciones de la multiplicación de matrices con las que creemos, cubrimos un amplio abanico de posibilidades teniendo en cuenta la arquitectura de la GPU. Para facilitar el análisis vamos a identificar cada una de ellas. En primer lugar, la implementación *tiling*, que mezcla las primeras optimizaciones del capítulo anterior: el tiling sobre memoria compartida y el tiling sobre el area de C a calcular. Aparecen en esta implementación los tres parámetros TN, TM y TL explicados anteriormente

(nótese que los casos en los que $by = TN$ y $bx = TM$ el tiling se realiza únicamente sobre memoria compartida y cada hilo calcula un único elemento de la matriz C). Además, esta implementación utiliza memoria local (arrays) para almacenar los resultados intermedios siempre que $by < TN$ o $bx < TM$ (en cuyo caso, cada hilo calcula varios elementos de la matriz C). La implementación *tiling_regs* es igual que la anterior pero utilizando siempre registros en lugar de memoria local. Por último, las implementaciones *todo_a* y *todo_b* se refieren la última optimización, en la que se lleva todo un bloque de la matriz A y la mitad de B y viceversa. Además, hemos compilado estas implementaciones de cuatro formas distintas: sin modificaciones, haciendo unrolling de los bucles internos, limitando el número de registros a 16; y combinado el límite de registros con unrolling.

Para realizar la exploración, hemos considerado matrices A de tamaño 1024x128, matrices B de 128x1024 y matrices C de 1024x1024 todas de tipo punto flotante. Hemos variado la configuración de los kernel como sigue:

- BY y BX (número de hilos en cada dimensión del bloque de hilos) han tomado los valores 8, 16, 32 y 64. Respetando siempre $BY * BX <= 512$
- TN, TM y TL (parámetros dedicados al tiling) han tomado los valores 8, 16, 32, 64, 96, 128, 192 y 256. Respetando siempre las condiciones de memoria compartida $(TN + TM) * TL * 4 <= 16384$ y $(TN + TM/2) * TL * 4 <= 16384$, $(TN/2 + TM) * TL * 4 <= 16384$ en las implementaciones *todo_a* y *todo_b*

La Figura 5.1 muestra los tiempos de ejecución obtenidos en función del Occupancy, de todas las implementaciones. La tendencia de los mínimos y de la media es disminuir a medida que aumenta el Occupancy hasta 0.5; y contra todo pronóstico, ya que aumentar el Occupancy equivale a aumentar el paralelismo, seguir aumentando el Occupancy penaliza el rendimiento. Esto refleja que aumentar el paralelismo para ocultar las latencias a memoria sin ningún tipo de control no es una buena estrategia. Además, podemos ver que la mayor densidad de soluciones se encuentran para los valores de Occupancy de 0.5 e inferiores. De hecho, muchas de estas soluciones se encuentran entre los mejores resultados en cuanto a rendimiento.

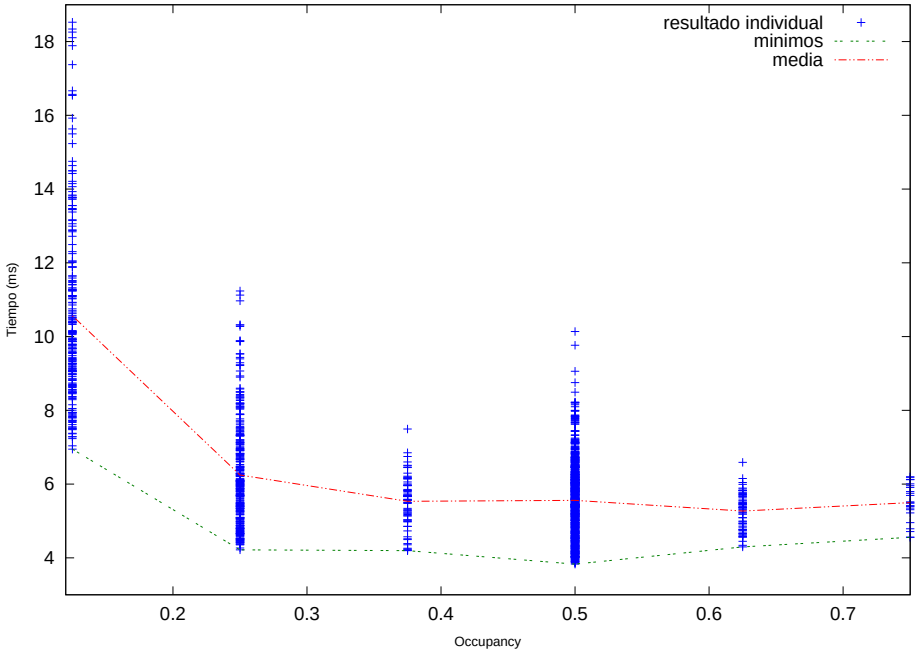


Figura 5.1: Variación del tiempo de ejecución en función del occupancy

Por otro lado, NVIDIA también recomienda configurar los kernels de forma que los bloques sean pequeños y al menos tres o más bloques estén activos en cada multiprocesador; de forma que se puedan ocultar latencias debidas a barreras de sincronización entre hilos del mismo bloque. En la Figura 5.2 muestramos la variación del tiempo de ejecución frente al número de bloques activos por multiprocesador.

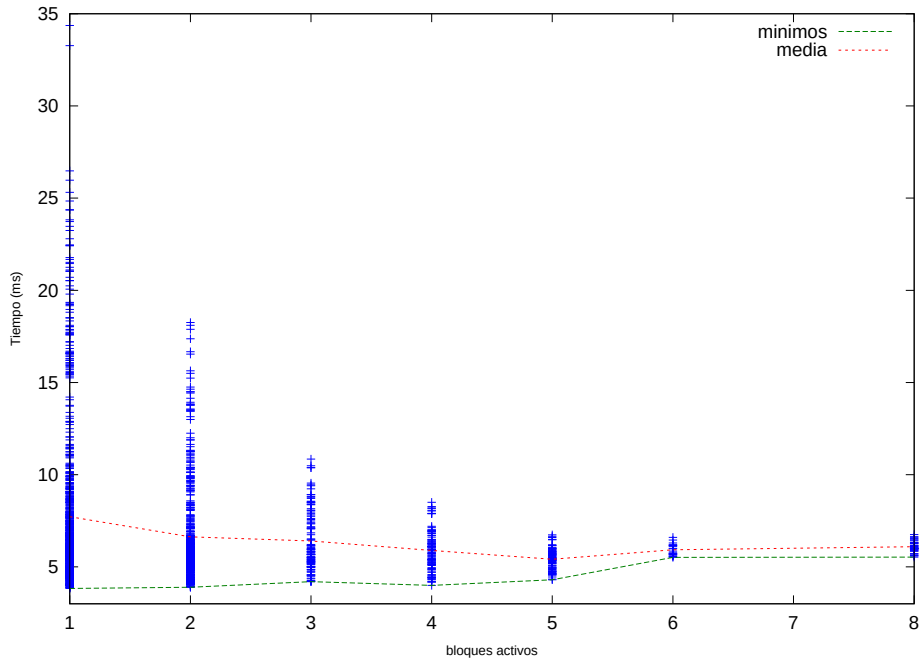


Figura 5.2: Variación del tiempo de ejecución en función del número de bloques activos

Aunque parece que la media de los tiempos tiende a disminuir a medida que el número de bloques activos aumenta, podemos ver que con los tiempos mínimos pasa todo lo contrario. Las mejores (y peores) soluciones se encuentran en las configuraciones que mantienen menos bloques activos. En esta aplicación en concreto, las recomendaciones que hace NVIDIA sobre el número de bloques no son las más acertadas. Aunque en la multiplicación de matrices existen barreras de sincronización en los bloques, es más importante poder asignar más memoria compartida a un bloque para aprovechar la localidad de los datos. Esto conlleva a tener menos bloques debido a las limitaciones de memoria compartida. La Figura 5.2, junto con la Figura 5.1, nos sirve para plantear la hipótesis de que: a priori el Occupancy no es la métrica más relevante, de hecho es una consecuencia de otras decisiones de configuración. Por otro lado, la correcta explotación del sistema de memoria de la GPU a costa de sacrificar paralelismo es el camino correcto hacia las soluciones óptimas.

En el resto del capítulo trateremos de explorar la correcta explotación del sistema de memoria en las implementaciones de la multiplicación de matrices, y veremos si los resultados se refuerzan con los resultados de las convoluciones. Al decidir que un bloque de hilos resuelve un tile de la matriz C, se producen copias de los datos en las distintas regiones de memoria asignadas a los bloques. Bloques de hilos que procesen los elementos

correspondientes a las mismas filas de C mantienen los mismos datos de la matriz A en sus memorias compartidas; lo mismo ocurre con los bloques de hilos que procesen elementos de las mismas columnas de C, cada bloque mantiene las mismas copias de la matriz B en su memoria compartida. Por lo tanto a mayor número de bloques de hilos totales, mayor número de copias en la memoria compartida y peor reuso de los datos. Para ilustrar esto vamos a definir la métrica *duplicidad* como la suma del número de tiles a lo ancho y largo en que se divide la matriz C; esto es, el número de copias en las memorias compartidas.

$$duplicidad = N/TN + M/TM \tag{5.1}$$

La Figura 5.3 muestra el tiempo de ejecución frente a la duplicidad. Separa por colores las distintas implementaciones además de separar la implementación *tiling* en dos distintas en función del trabajo asignado a cada hilo. Por la propia naturaleza de las implementaciones *todo_a* y *todo_b*, cada hilo calcula al menos dos elementos. Vemos como las implementaciones en las que un hilo calcula un único elemento se situan a la derecha de la gráfica con valores de duplicidad muy altos. Al calcular un único elemento se cumple que $BY = TN$ y $BX = TM$; como los bloques no pueden tener más de 512 hilos BX y BY son muy pequeños y provocan que se lancen muchos más bloques, lo que implica mucha más duplicidad. Sin embargo, las implementaciones en las que un hilo calcula más de un elemento permiten valores de TN y TM mucho más grandes, ahora sólo límitados por la cantidad de memoria compartida asignada a cada bloque. Por eso, estas implementaciones se encuentran en la parte izquierda de la gráfica.

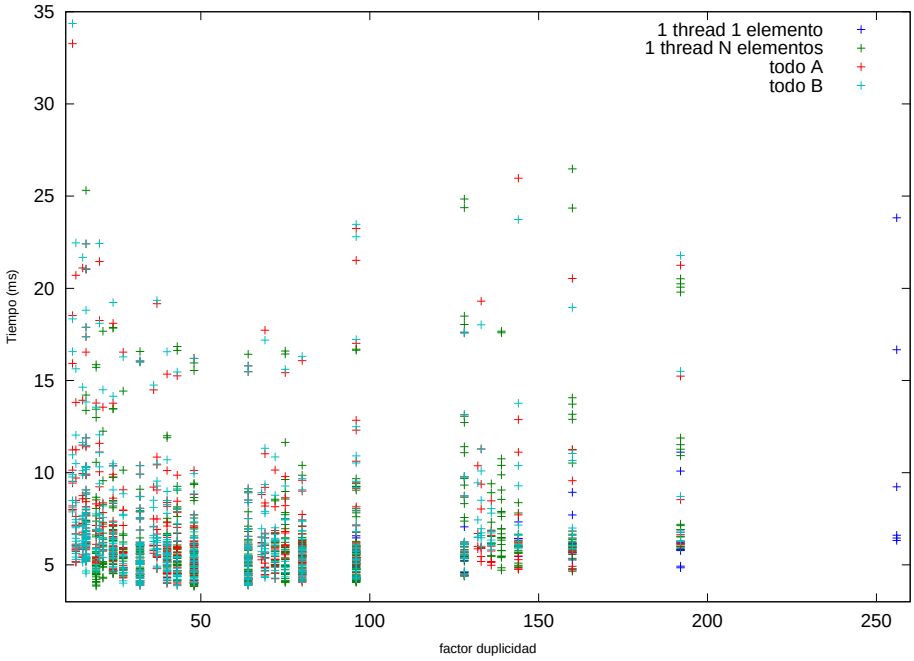


Figura 5.3: Variación del tiempo de ejecución en función del número de la duplicidad

Aún con todo esto, la gráfica muestra soluciones lentas en todos los rangos de duplicidad y no se vé una tendencia ascendente clara que demuestre que a mayor duplicidad peor rendimiento. Esto se debe a que la duplicidad no es el único factor que afecta al rendimiento.

En el Capítulo 3 se explicó que, bloques de hilos cuyo ancho no sea multiplo de 16 y el patrón de acceso sea tal que hilos consecutivos acceden a datos consecutivos, provoca que todos los accesos sean no coalesced. Esto se ratifica con los resultados del profiler, en los que podemos ver que todos las configuraciones cuyo ancho (BX) es 8, realizan sus accesos a memoria en peticiones de 32B en lugar de 64B. Esto duplica el número de accesos y divide el ancho de banda efectivo de la memoria global. Además, dado que los tiles que se llevan a memoria compartida tienen un ancho que es siempre multiplo de 16, todos las configuraciones definidas con $BX = 8$ también provocan conflictos en el acceso a la memoria compartida. Esto también se demuestra en el profiler mediante el contador *warp_serialize*.

Siguiendo con la hipótesis de explotar lo mejor posible el sistema de memoria, vamos a ir elimimando soluciones que claramente no haga un buen uso del sistema de memoria. La Figura 5.4 muestra de nuevo el tiempo de ejecución en función de la duplicidad. Sin

embargo, hemos eliminado casi todas las implementaciones cuyos accesos no eran coalesced y aquellos que producían conflictos en los accesos a memoria compartida ($BX = 8$). Podemos observar que cuando la duplicidad es demasiado pequeña, menor de 20, los tiempos son más elevados. Esto se debe a que los tiles son tan grandes que se ha perdido mucho paralelismo. Además, al ser TN y TM tan grandes, el rango en el que se puede mover TL es muy pequeño (debido a las limitaciones de memoria compartida) lo que obliga a que el algoritmo se desarrolle en muchos pasos. En un rango de duplicidad entre 20 y 100, los tiempos se mantienen en el mismo rango, se sitúa la mayor densidad de soluciones y se encuentran los tiempos óptimos. En este rango, TL tiene más libertad y se reduce la complejidad del algoritmo. De ahí en adelante, aparece una ligera tendencia ascendente en el tiempo de ejecución a medida que aumenta la duplicidad.

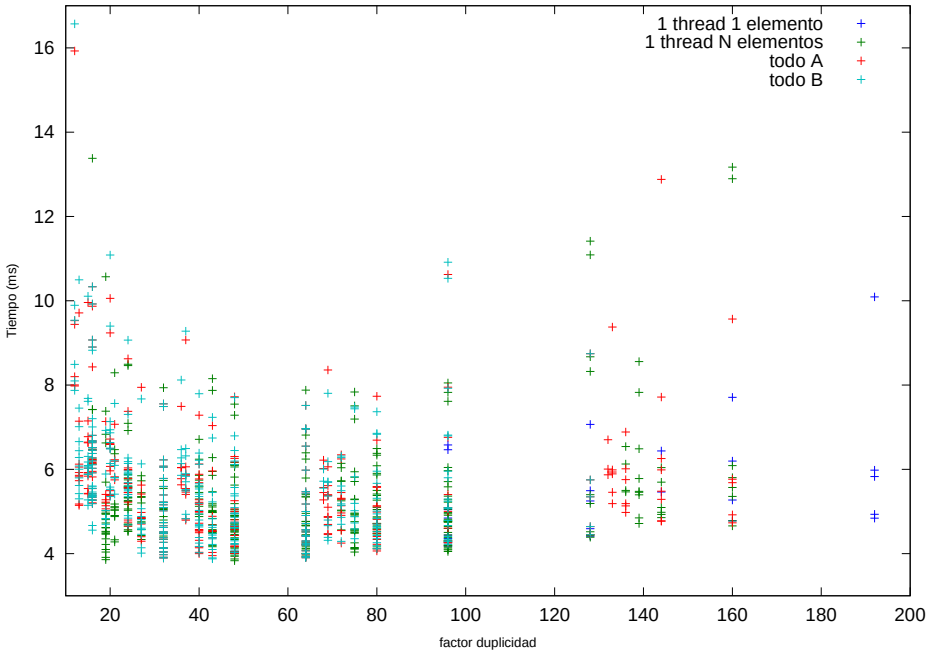


Figura 5.4: Variación del tiempo de ejecución en función del número de la duplicidad con accesos coalesced y sin conflictos en memoria compartida

En la gráfica anterior hemos tratado de eliminar todas las soluciones que realizaban accesos a memoria no coalesced exigiendo que BX fuese multiplo de 16. Sin embargo, no es cierto que se hayan elimnado todas. La relación entre los parámetros BX y TL puede dar lugar a lecturas no coalesced al llevar el trozo de A a la memoria compartida. Recuérdese que el bloque de A que se lleva a memoria compartida es de $TN*TL$. Entonces,

si $BX > TL$ en primer lugar habrá hilos parados en la fase de carga. Esto equivale a divergencias en los saltos, ya que los hilos deben comprobar si tienen que leer el dato de memoria o no. Además aplicando el mismo razonamiento que antes, si TL no es multiplo de 16, la carga del bloque A se realizará con lecturas no coalesced. De nuevo, esto se verifica en el profiler al ver que algunas de las lecturas se realizan como peticiones de 32B.

Por otro lado, TL tiene otro impacto importante sobre el número de instrucciones dinámicas ejecutadas, ya que controla el número de pasos (L/TL) que necesita cada bloque para completar su cómputo; es decir, controla el bucle externo que ejecutan todos los hilos. Por tanto a menor número de iteraciones, lo que se refleja con un TL elevado, menor número de instrucciones dinámicas. Esta evolución en el número de instrucciones dinámicas puede verse en la siguiente gráfica.

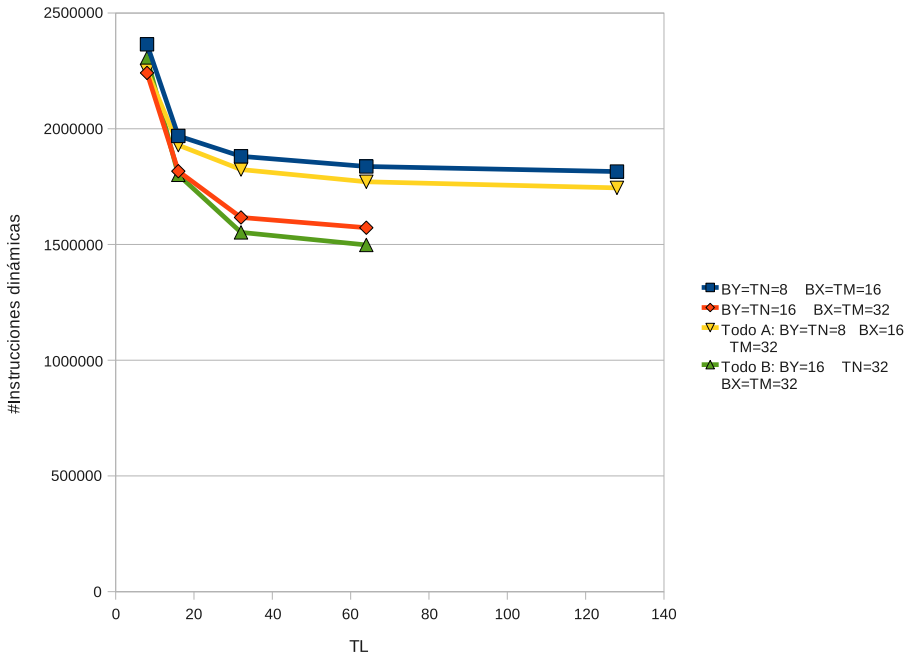


Figura 5.5: Cómo afecta TL a las instrucciones dinámicas

En la Figura 5.5 vemos que, efectivamente, aumentar el valor de TL tiene una importante repercusión en el número de instrucciones dinámicas. Además, podemos diferenciar dos partes en la gráfica. Mientras TL es menor que BX, los cambios en el número de instrucciones dinámicas son mucho más pronunciados; sin embargo, una vez se cumple que $TL >= BX$, las instrucciones siguen disminuyendo pero con una pendiente mucho menos

pronunciada. Esto se debe en parte, a lo que hemos comentado acerca de las divergencias en los saltos, en cuyo caso los warps deben ejecutar ambas ramas de forma secuencial. Esto aumenta el número de instrucciones dinámicas, así como los accesos no coalesced también aumentan el número de instrucciones.

La Figura 5.6 muestra de nuevo la relación entre el tiempo y la duplicidad, pero eliminando definitivamente todos los accesos no coalesced y todas las posibles divergencias en los saltos debidas al parámetro TL.

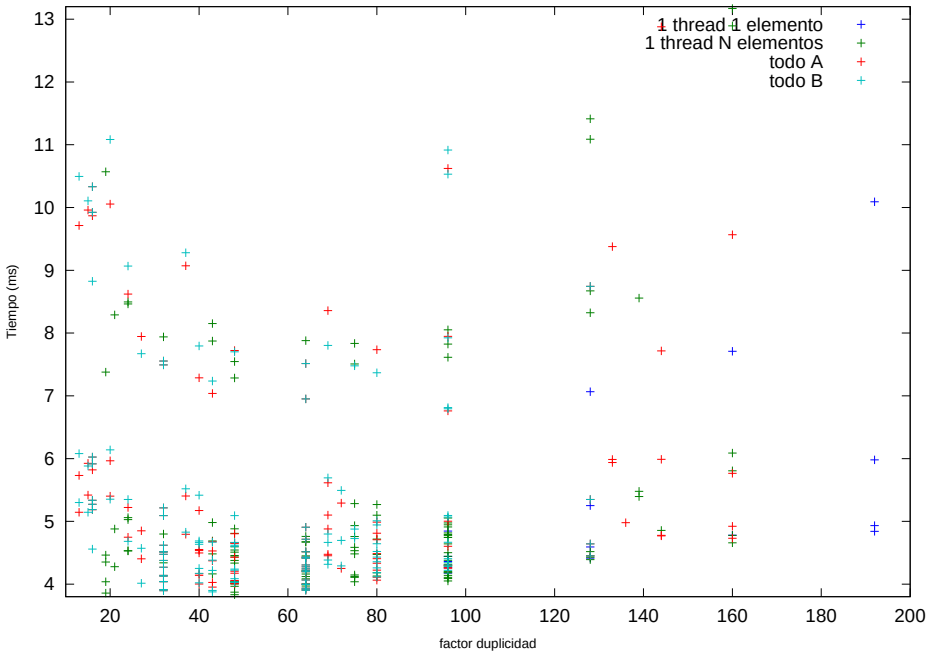


Figura 5.6: Reducir instrucciones y evitar saltos divergentes

Hemos eliminado muchos resultados con tiempos entre 5ms y 8ms a lo largo de todo el rango de duplicidad; es decir, como hemos eliminado soluciones que incrementaban el número de instrucciones y dichas soluciones estaban a lo largo de toda la duplicidad, podemos ver que no existe una relación entre ambas métricas. Asique tomaremos el número de instrucciones como otra métrica a analizar. En los resultados restantes, aparece un hueco vacío por el cual se salta de soluciones con tiempos buenos a otras con peores resultados. Ahora que los accesos a memoria ya están optimizados, esta perdida en el rendimiento se debe en parte al sacrificio excesivo de paralelismo. Hemos visto hasta ahora que menor duplicidad era sinónimo de falta de paralelismo debido a que se lanzan menos bloques de hilos; sin embargo no es la única forma de abordar el paralelismo. Vamos a usar el

Occupancy como medida para filtrar las soluciones que no aprovechan en un porcentaje alto el multiprocesador.

La Figura 5.8 muestra la duplicidad frente el tiempo después de haber eliminado todos los resultados que no aprovechasen al menos la mitad de los recursos de los multiprocesadores. Es importante notar que la escala de tiempo de los resultados mostrados es casi la tercera parte que la mostrada en la Figura 5.6. Vemos que la tendencia según aumenta la duplicidad ya es claramente creciente; salvo para las duplicidades más pequeñas en las que se lanzan menos bloques y, aunque ahora tenemos la certeza que son bloques con muchos hilos, hacen uso de mucha memoria local por hilo para mantener los resultados intermedios. Además, estas soluciones utlizan valores de TL pequeños de forma que el número de pasos para realizar el cómputo aumenta, aumentando el número de instrucciones dinámicas y el número de lecturas y escrituras de esa memoria local.

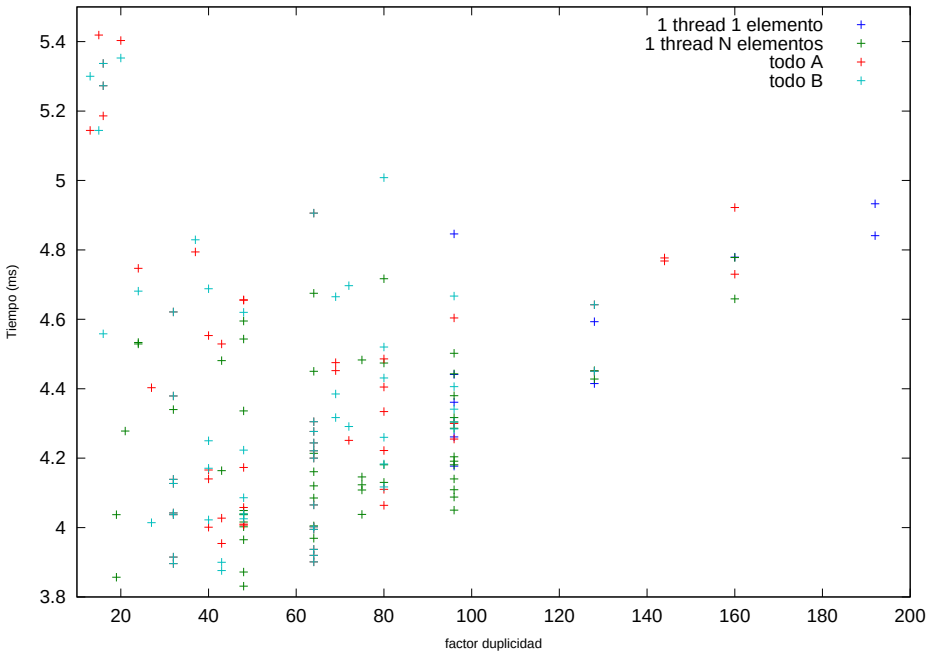


Figura 5.7: Maximizar paralelismo de los restantes (Duplicidad)

Como hemos visto, el número de instrucciones dinámicas es una métrica que se relaciona de forma directamente proporcional al tiempo de ejecución de forma que podemos expresar los resultados anteriores en función de las instrucciones ejecutadas. A continuación se muestra dicha relación.

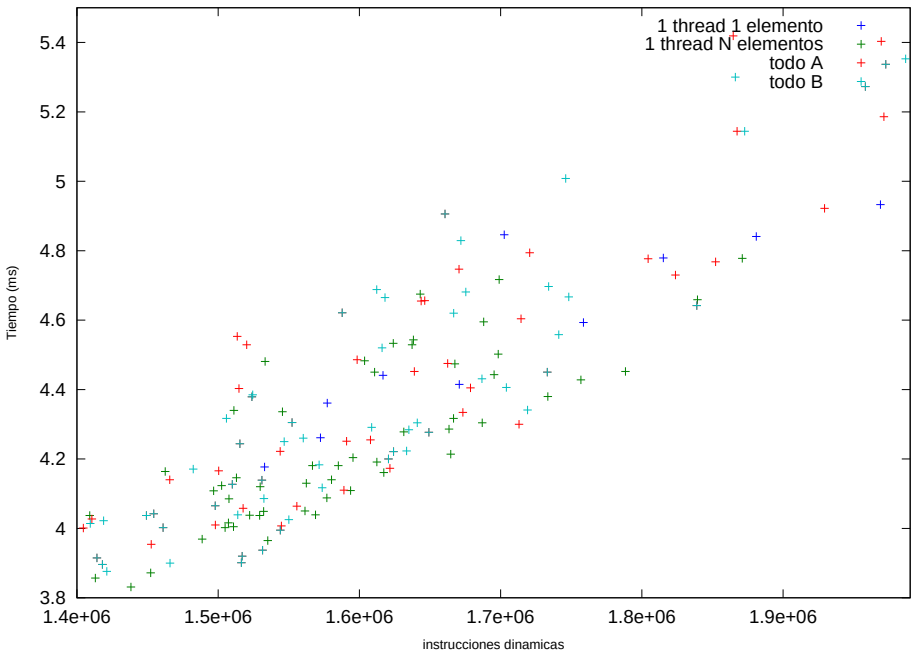


Figura 5.8: Maximizar paralelismo de los restantes (Instrucciones)

Con este proceso de filtrado hemos conseguido reducir el espacio de búsqueda de forma que casi es posible realizar un búsqueda exhaustiva, en la que incluso se podría llegar a ejecutar cada una de las implementaciones para seleccionar el óptimo. Además, poder estimar el número de instrucciones dinámicas sería de gran utilidad para poder tomar decisiones a la hora de buscar soluciones óptimas.

Llegados a este punto hemos explotado el sistema de memoria de forma eficiente, pero a lo largo de este trabajo se ha hablado de otras técnicas para mejorar el rendimiento. En este capítulo también hemos analizado los efectos de estas técnicas. En el Capítulo 3 hablamos del unrolling con estrategia para aumentar el ILP y reducir el número de instrucciones dinámicas. Teniendo en cuenta que las instrucciones dinámicas están fuertemente relacionadas con el tiempo de ejecución, disminuirlas es una buena manera de mejorar el rendimiento. Hemos desenrrollado los tres bucles más internos (en caso de haberlos). En las implementaciones en las que un hilo calcula varios elementos, se ha desenrrollado los bucles de carga en memoria compartida y el bucle del producto escalar; mientras cuando un hilo calcula un único elemento solo se ha desenrrollado el bucle del producto escalar. La Figura 5.9 muestra los efectos del unrolling sobre las quince mejores implementaciones, aunque el efecto es similar para todos los casos.

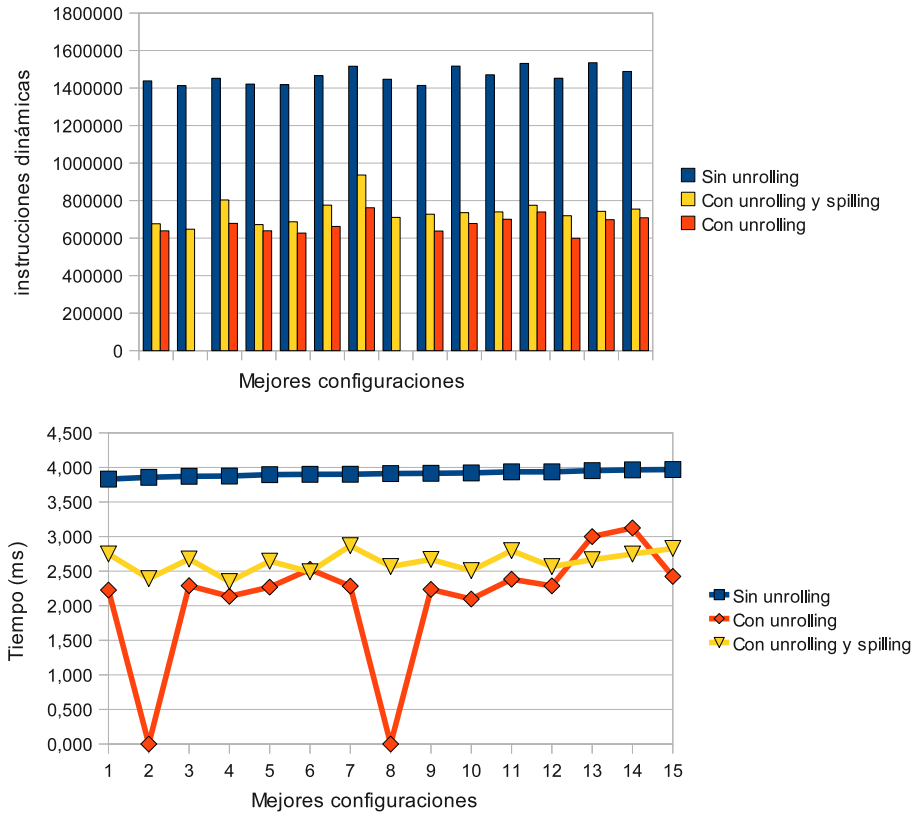


Figura 5.9: Instrucciones dinámicas y tiempo de ejecución en función del unrolling

En el caso del unrolling, el número de instrucciones se reduce aproximadamente a la mitad salvo en la configuración dos y ocho, en las que el unrolling eleva tanto el número de registros a usar por hilo, que el kernel no se puede ejecutar. La barra amarilla muestra el efecto de combinar unrolling y spilling. El número de instrucciones es ligeramente mayor que las de unrolling debido a los accesos a memoria local. El tiempo de ejecución también se reduce casi a la mitad de la versión sin unrolling a la versión con unrolling. Aunque al hacer unrolling todas las soluciones mejoran, no mejoran en la misma proporción; por esto, la gráfica de tiempos del unrolling no es creciente.

Ya hemos hablado a lo largo del trabajo de la limitación de Occupancy por culpa de un uso alto de registros por hilo. En el Capítulo 3 hablamos de cómo los registros podían limitar el paralelismo y de cómo forzar que un kernel use menos registros por hilo. El Cuadro 5.2 muestra la cantidad de registros que utilizan nuestras implementaciones de la

multiplicación de matrices.

Implementación	**Registros**
tiling (1 hilo 1 elemento)	20
tiling (1 hilo N elemento)	20-23
tiling_reg	24-27
todo_a	20-24
todo_b	20-24

Cuadro 5.2: Número de registros que utiliza cada implementación

Las implementaciones en las que un hilo calcula varios resultados (*tiling (1 hilo N elemento)*, *tiling_reg*, *todo_a* y *todo_b*) varían el número de registros que utilizan dependiendo de cúantos elementos calcule cada hilo. De forma que, si hilo calcula más elementos, utiliza más registros. Además, *tiling_reg* utiliza más registros porque evita el uso de memoria local forzando a que los resultados intermedios se alojen en registros.

Las siguientes gráficas muestran el efecto de forzar el número de registros por hilo a 16. Recuérdese que 16 registros por hilo es el máximo número de registros permidos para poder aprovechar todos los recursos de los multiprocesadores (alcanzar el 100 % de Occupancy). Las gráficas muestran speedup frente a la mejora del Occupancy debido al menor uso de registros. Para estas gráficas, sólo se han tenido en cuenta las implementaciones que usan entre 2K y 4K de memoria compartida por bloque; con el objetivo de que la memoria compartida no limite el posible aumento de Occupancy.

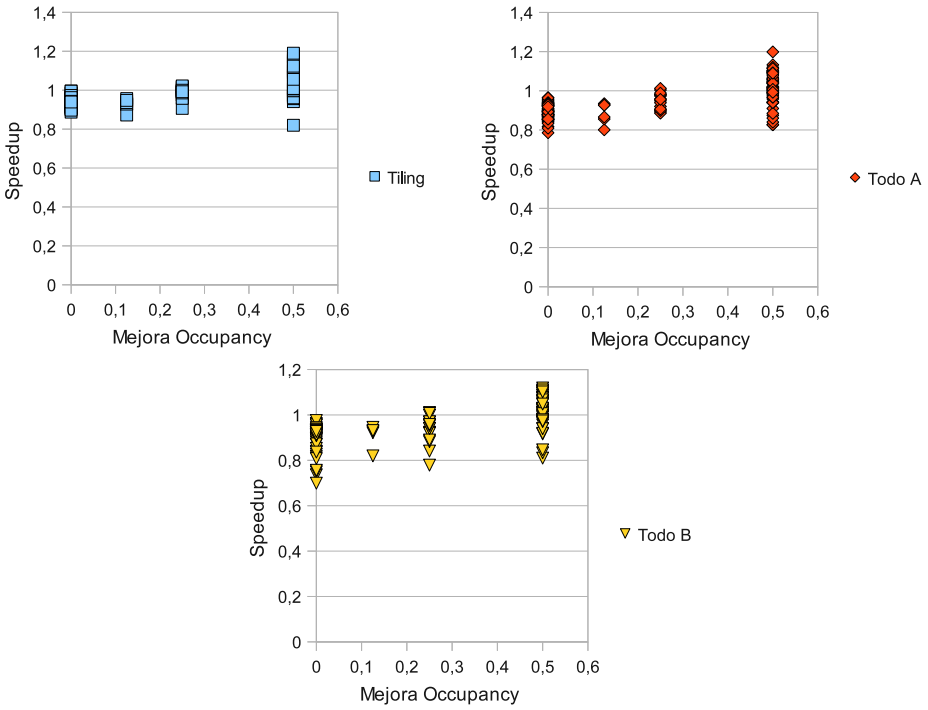


Figura 5.10: Mejora del rendimiento en función de la mejora de Occupancy al hacer spilling

En la Figura 5.10 vemos que, para que la penalización que produce el spilling en el rendimiento empiece a compensar, el Occupancy debe mejorar al menos en 0.25 (la cuarta parte de los recursos del multiprocesador); y aún así, la mejora no está garantizada. Sin embargo, al combinar spilling con unrolling el efecto sobre el rendimiento es mucho más efectiva y en los casos que conseguimos aumentar el Occupancy, el uso de memoria local (debido al spilling) se oculta con los beneficios del unrolling. A continuación, la Figura 5.11 muestra los efectos de combinar ambas técnicas.

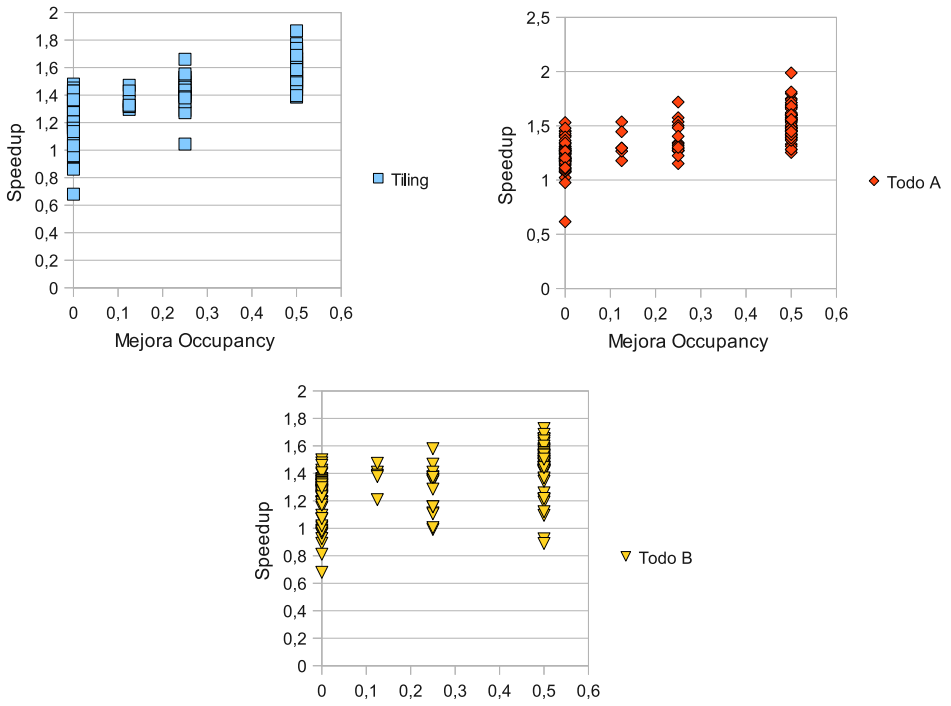


Figura 5.11: Mejora del rendimiento en función de la mejora de Occupancy al hacer spilling y unrolling

Hasta ahora, hemos considerado tamaños de problema (1024x1024) que siempre generaban bloques suficientes para llenar todos los multiprocesadores. Hemos comprobado qué ocurre con tamaños de problema menores (matriz C de tamaño 512x512). El segundo mejor resultado para tamaños grandes divide C en bloques de 128x96. Este tamaño de tile, para una matriz de 512x512, genera únicamente 24 bloques en total. Este número de bloques es insuficiente para asignar trabajo a todos los multiprocesadores (30 en la tarjeta C1060). Aunque aún obtiene buenos resultados, pues aún utiliza el 80 % de los multiprocesadores, ya no forma parte de los mejores resultados; ahora otras soluciones que no sacrifican tanto paralelismo (tiles de 32x32 ó 32x64) obtienen resultados más efectivos.

Los resultados obtenidos de la convolución verifican el análisis realizado sobre los resultados de la multiplicación de matrices. Hemos seguido el mismo proceso de filtrado, empezando por mejorar el acceso al sistema de memoria. Sin embargo, los kernels de la convolución son mucho más sencillos que la multiplicación. El concepto de duplicidad pierde peso en las decisiones, ya que las copias que hay en las distintas memorias compartidas

dependen del tamaño de la máscara de convolución.

$$duplicidad(kernelfilas) = 2 * RAD\_MASK * (M/TM) \tag{5.2}$$

$$duplicidad(kernelcolumnas) = 2 * RAD\_MASK * (N/TN) \tag{5.3}$$

De nuevo, a medida que el tamaño de los tiles es más grande, la duplicidad es más pequeña; pero como cada bloque utiliza muy poca memoria compartida, ésta no supone una limitación como era el caso de la multiplicación. Esto, combinado con un número de registros muy bajo por hilo (12), deja un amplio margen para alcanzar el Occupancy máximo.

Capítulo 6

Conclusiones y trabajo futuro

La complejidad de un solo core ha llegado a tal punto que no es posible sacar más rendimiento con variaciones arquitectónicas. Por tanto, ya no puede esperarse que un mismo binario incremente sus prestaciones automáticamente con cada familia de procesadores. Los fabricantes, al no saber qué hacer con el área del que disponen, han apostado claramente hacia la replicación de cores. En ese contexto, las GPUs han emergido como plataformas de procesamiento masivamente paralelas. Si bien no son de proposito general, hay una gran variedad de aplicaciones que pueden beneficiarse de sus características. Para ello, el programador debe adaptarse a un nuevo paradigma de programación que requiere un importante cambio de perspectiva. Esto provoca que un desarrollador inexperto deba invertir tiempo en formarse y adaptarse a la nueva tecnología. Por ello, la utilización de herramientas de traducción guiadas o automáticas pueden facilitar este proceso.

Este tipo de herramientas deben ser capaces de llegar a soluciones óptimas. Para ello, deben saber cómo explotar los recursos disponibles en la GPU, entre ellos cabe destacar el sistema heterogéneo de memoria y su alta latencia, el diseño masivamente paralelo del procesador, en el que miles de hilos pueden colaborar con un objetivo común, optmizaciones sobre el flujo de instrucciones y el paralelismo a nivel de instrucción.

A lo largo de nuestra exploración, hemos querido ver cómo influyen determinadas decisiones de alto nivel en el rendimiento de la aplicación. A partir de esa información, pretendemos desarrollar una metodología que fije el orden en el que deben tomarse dichas decisiones y los criterios más relevantes que deben tenerse en cuenta. Las decisiones más relevantes son:

- Estudio del reuso/duplicidad de los datos, tanto inter-bloque como intra-bloque.
- Cantidad de memoria compartida asignada a cada bloque.
- Definir la granularidad de paralelismo.
- Número de bloques totales a ejecutar y número de hilos por bloque.
- Planificación de accesos al sistema de memoria.
- Geometría del bloque.

Aparentemente el orden en que deben tomarse estas decisiones debe ser tal que se prioricen aquellas que involucran al sistema de memoria, incluso sacrificando paralelismo si fuera necesario. De nuestra exploración manual hemos sacado las siguientes conclusiones:

- Si se explota correctamente el sistema de memoria no importa un menor número de bloques activos. Este sacrificio de paralelismo por supuesto tiene límites.
- Se debe evitar, en la medida de lo posible el uso de memoria local. Es conveniente duplicar el código estático si con ello se eliminan arrays.
- El ancho del bloque de hilos debe ser al menos 16 o un multiplo.
- Como mínimo se debe alcanzar un Occupancy del 25 % para ocultar latencias. Normalmente, las soluciones óptimas se encuentran con un Occupancy del 50 %. Forzar el aumento de Occupancy de forma artificial puede no afectar al rendimiento.
- Si no supone mucho coste en memoria y en complejidad, se debe aumentar la carga de trabajo de los hilos.
- Estrategias como unrolling y spilling son útiles para mejorar el rendimiento una vez se ha optimizado el resto del sistema.

Siguiendo esta línea de investigación, el trabajo futuro se basará inicialmente en llevar este estudio a un mayor número de aplicaciones. Tratando que el grado de similitud entre ellas sea mínimo. Mediante herramientas como GPGPU-Sim u Ocelot, trateremos de examinar el comportamiento real a bajo nivel; y así, probar distintas transformaciones sobre el código a alto nivel con herramientas de ayuda. Con el fin de obtener una metodología capaz de guiarnos a la hora de realizar traducciones, incluso capaz de definir los pasos exactos, y en el orden correcto, para realizar la traducción de aplicaciones de forma automática.

Apéndice A

Algoritmo NMF supervisado

La factorización no negativa de matrices (NMF) descompone un matriz X en un producto de dos matrices W y H, a las que se impone la restricción de que todos sus elementos deben ser mayor o igual que cero (no negativos). Como resultado de la descomposición, cada columna de X queda representada como una combinación lineal de columnas de W, dónde los coeficientes de la combinación son los elemnetos de la columna de H correspondiente. Se dice que las columnas de W representan una nueva base, cada una identifica una característica diferenciadora de los datos originales, que facilita su posterior clasificación. En la versión simple, el algoritmo no tiene ninguna guía para buscar los elementos de W. En la versión supervisada podemos añadir información al algoritmo para guiarlo, estableciendo como entrada una relación de las columnas de X que pertenecen a una misma clase.

Este tipo de algoritmos son utilizados frecuentemente en aplicaciones de reconocimiento de patrones y minería de datos. Por ejemplo, se ha utilizado NMF para la identificación de notas en piezas musicales, identificación de genes que intervienen en distintas patologías clínicas, reconocimiento facial, etc.

A continuación se muestra el pseudocódigo del algoritmo en MATLAB:

```
function [W, H, obj] = nmf_sup(V, W0, H0, classes, rank,
                               regularization, max_iter, thresold)

 [n m] = size(V);
 r= rank;
```

```
if isempty(W0)
   W = rand(n,r);
   W = W ./repmat(sum(W,1),n,1));
   H = rand(r,m);
else
   W = W0;
   H = H0;
end

C = lenght(distinctClasses);

for i=1:C
  idxClasses{i} = find(classes==distinctClasses(i));
  Vclass{i} = V(:,idxClasses{i});
  Hclass{i} = H(:,idxClasses{i});
end

[obj_old o1 o2] = compute_objetive(V, Hclass, W, H, regularization);
obj = obj_old;

for k=1:max_iter
  obj_old = obj;

  W = W .* (V*H' ./ W*H*H') ./ repmat( sum(W,1), n, 1);

  [obj_medium o1 o2 ] = compute_objetive(V, Hclass, W, H, regularization);
  if obj_old < obj_medium
    obj_old = obj_medium;
  end

  H = linearSearchH(V, idxClasses, Vclass, Hclass, W, H, regularization);

  for i=1:C
    Hclass{i} = H(:,idxClasses{i});
  end
```

```
   [obj o1 o2] = compute_objetive(V, Hclass, W, H, regularization);
   if obj_old - obj < thresold
     return
   end
 end
end



function newH = linearSearchH(V, idxClasses, Vclass, Hclass,
                              W, H, regularization)
 newH = H;
 C=lenght(Vclass);

 gradientH = zeros(size(H));

 for i=1:C
   AUX = zeros(size(Hclass{i});
   for j=1:C
     if i==j continue; end
     AUX = AUX + Hclass{j}*Hclass{j}'*Hclass{i};
   end
   gradientHi = -2*W'*(Vclass{i} - W*Hclass{i}) + 2*regularization*AUX;
   gradientH(:,idxClasses{i}) = gradientHi;
 end

 Haux = H;
 Hclassaux;

 obj_old = compute_objetive(V,Hclass,W,H,regularization);
 candidate_lambda = 0.5*H ./ (W'*W*H);

 lambda_max = max(candidate_lambda(candidate_lambda>0));
 lambda_min = min(candidate_lambda(candidate_lambda>0));
 if lenght(lambdamin)==0
```

```
    lambda = 1e-7;
  else
    lambda = lambda_min;
  end

  Haux = H - lambda*gradientH;
  Haux(Haux<0) = 0;
  for i=1:C
    Hauxclass{i} = Haux(:,idxClasses{i});
  end

  obj_new = compute_objetive(V, Hclassaux, W, Haux, regularization);
  if obj_old > obj_new
    newH = H;
    obj_old = obj_new;
  end

  while lambda<lambda_max
    lambda = lambda*2;
    Haux = H - lambda*gradientH;
    Haux(Haux<0) = 0;

    for i=1:C
      Hauxclass{i} = Haux(:,idxClasses{i});
    end

    obj_new = compute_objetive(V, Hclassaux, W, Haux, regularization);
    if obj_old > obj_new
      newH = H;
      obj_old = obj_new;
    else
       break;
    end
  end
end
```

```
function [obj o1 o2] = compute_objetive(V, Hclass,
                                        W, H, regularization)
 V_WH = V - W*H;
 o1 = sum(V_WH(:).^2);

 C=lenght(Hclass);
 for i=1:C
   for j=1:C
     if i==j continue; end
     HitHj = Hclass{i}'*Hclass{j};
     o2 = o2 + sum(HitHj(:).^2);
   end
 end
 obj = o1 + regularization*o2;
end
```

Como puede verse en el código, hay definidas tres funciones:

- La primera mantiene el bucle principal del algoritmo y calcula la matriz W en cada interación.
- La segunda corresponde al cálculo de la matriz H mediante un proceso iterativo.
- La tercera comprueba si se han alcanzado los objetivos.

En las tres funciones anteriores se hace un uso intensivo de cálculo matricial lo que convierte a este algoritmo en idóneo para ejecutarlo en la GPU. La traducción a GPU se ha realizado llevando a cabo la traducción de operaciones elementales a kernels. Las operaciones punto a punto se han configurado para maximizar el Occupancy ya que no hace uso de memoria compartida. Para la multiplicación de matrices se ha usado el kernel estudiado a lo largo de este trabajo. Finalmente, para la reducción de matrices se ha implementado un kernel en el que los hilos hacen la reducción de forma lineal en lugar de logarítmica; ya que la versión logarítmica para matrices grandes requiere muchas escrituras en memoria global entre las iteraciones además de ir dejando poco a poco hilos sin trabajo.

Se han minimizado las comunicaciones entre CPU y GPU. Hemos asumido que tenemos memoria suficiente en la tarjeta para alojar todos los datos (Recuérdese que la tarjeta

C1060 cuenta con 4GB de memoria global), así las matrices V, W y H se han cargado al inicio del algoritmo y los resultados se han descargado un vez finalizado sin tener que realizar comunicaciones adicionales.

Las pruebas se han realizado con matrices V de 16384x480 con 32 clases ($rank = 32$). Se han conseguido speedups de hasta 12 frente a la implementación en CPU[1].

[1]Para un mismo número de itaraciones

Bibliografía

[1] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen mei W. Hwu. An adaptative performance modeling tool for gpu architectures. *PPoPP'10, January 9-14*, 2010.

[2] José María Cecilia, José Manuel García, and Manuel Ujaldón. The gpu on the matrix-matrix multiply: Performance study and contributions. 2010.

[3] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *ISCA'09, June 20-24*, 2009.

[4] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. Modeling gpu-cpu workloads and systems. *GPGPU-3 March 14 2010*, 2010.

[5] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 1 edition, 2010.

[6] NVIDIA Corporation. *NVIDIA CUDA Best practices guide 3.0*, 2010.

[7] NVIDIA Corporation. *NVIDIA CUDA Programming guide 3.0*, 2010.

[8] NVIDIA Corporation. *PTX: Parallel Thread Execution ISA 2.0*, 2010.

[9] NVIDIA technical team. Nvidia geforce 8800 gpu architecture overview. Technical report, NVIDIA Corporation, 2006.

[10] NVIDIA technical team. Nvidia geforce gtx 200 gpu architecture overview. Technical report, NVIDIA Corporation, 2008.

[11] NVIDIA technical team. Nvidia fermi whitepaper. Technical report, NVIDIA Corporation, 2010.